Développement d'outils d'optimisation pour freefem++

Sylvain Auliac

Université Pierre et Marie Curie - Paris 6

Laboratoire Jacques-Louis Lions - UMR 7598

THÈSE

PRÉSENTÉE A

L'UNIVERSITÉ PIERRE ET MARIE CURIE

ÉCOLE DOCTORALE DE SCIENCES MATHEMATIQUES DE PARIS CENTRE

Par Sylvain Auliac

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : MATHEMATIQUES

Développement d'outils d'optimisation pour FreeFem++

Directeur de recherche : Frédéric Hecht

Soutenue le 11 mars 2014 devant le jury composé de :

Ionut DANAILA	Examinateur
Laurent DUMAS	Examinateur
Frédéric HECHT	Directeur
Bijan MOHAMMADI	Rapporteur
Olivier PIRONNEAU	Examinateur

Ecole Doctorale de Sciences Mathématiques de Paris Centre

UFR 929 - Mathématiques

Laboratoire Jacques-Louis Lions
4 place Jussieu
75 005 Paris

École doctorale Paris centre Case 188
4 place Jussieu
75 252 Paris cedex 05

Remerciements

J'aimerais tout d'abord remercier Frédéric Hecht sans qui, bien évidemment, cette thèse ne se serait tout simplement pas concrétisée. Mais au-delà du temps qu'il m'a consacré en tant que son disciple, et au risque de ne pas me montrer très original, c'est surtout pour son soutien, tant matériel que moral, que je lui suis reconnaissant, pour son aide providentielle dans les moments les plus critiques, ainsi que pour sa patience et son indulgence lors des périodes de doute que j'ai pu traverser pendant ces années.

La lecture d'un manuscrit de thèse, de surcroît une thèse de mathématiques, ne doit pas être une tâche particulièrement divertissante, aussi, je remercie André Fortin et Bijan Mohammadi, qui ont consenti à se constituer rapporteurs, et ont scrupuleusement commenté ce travail. Je souhaiterais aussi remercier Ionut Danaila, Laurent Dumas et Olivier Pironneau pour le temps qu'il m'ont consacré en acceptant de faire partie du jury pour ma soutenance, avec toutes les responsabilités que cela implique.

Grâce soit rendue à Mouna, pour ses relectures minutieuses, qui éviteront à ce manuscrit d'entrer dans les annales de la cacographie. Merci aussi de supporter au quotidien l'ours mal léché que je suis.

Merci à Yvon Maday et à Robert Longeon qui ont tous deux joué un rôle important dans le financement de la fin de ma thèse. Cette rémunération se sera révélée salutaire car, sans elle, le récent affermissement des règles administratives m'aurait condamné à ne jamais goûter au fruit de ces années de dur labeur. A ce propos, j'aimerais également remercier Gilles Godefroy pour sa bienveillance, et pour m'avoir opportunément aidé à éviter les conséquences fâcheuses que me réservait le protocole.

Merci à Jacques Périaux pour l'invitation en Finlande au cœur de l'hiver. Cela aura été l'un des voyages les plus dépaysants qu'il m'aura été donné de faire. Merci aussi à Faker Ben Belgacem pour m'avoir convié à participer à l'ICOSAHOM et ainsi permis de découvrir les charmes de la Tunisie, avec son terrible soleil et ses bricks au thon.

Le personnel des secrétariats du laboratoire et de l'école doctorale mérite sans aucun doute aussi une place ici : merci à eux pour leur sympathie et leur efficacité. Et à mes collègues du bureau 301, j'adresse tous mes encouragements pour finir leur thèse. Quant à ceux qui l'ont déjà terminée et s'en sont allés dans la nature, je souhaite que les vents leur soient favorables. Merci à Nicolas L., pour sa bonne humeur communicative.

Je tiens à adresser de sincères salutations à toutes les autres personnes avec qui j'ai pu entretenir des relations au cours de mon doctorat, que ce soit au laboratoire ou lors des événements scientifiques auxquels j'ai pris part.

Mille mercis à mes parents, pour s'être plutôt bien débrouillés avec nous trois. Merci à Pierrick pour le félin de Sibérie, et à Raphaël pour les pâtes multicolores.

Grand merci à Yael et François : si le hasard amenait un exemplaire de cette thèse dans vos mains, sachez que vous avez généreusement contribué à ce qu'elle s'achève dans la sérénité.

Merci aux amis de longue date, à Benjamin, Laurent, Matthieu, Ouanès, et Perceval.

Merci à Jean Hervé pour m'avoir extirpé de la catégorie des poids plumes.
Enfin, merci à ceux qui auront la patience de lire cette thèse.

Résumé

Résumé

Cette thèse est consacrée au développement d'outils pour FreeFem++ destinés à faciliter la résolution des problèmes d'optimisation. Ce travail se compose de deux parties principales. La première consiste en la programmation, la validation et l'exploitation d'interfaces permettant l'utilisation de routines d'optimisation directement dans le logiciel. La seconde comprend le développement de solutions pour le calcul automatisé des dérivées, toujours au sein de FreeFem++, en exploitant les paradigmes de la différentiation automatique.

FreeFem++ est un environnement de développement intégré dédié à la résolution numérique d'équations aux dérivées partielles en dimension 2 et 3. Son langage ergonomique permet à l'utilisateur d'exploiter aisément ses nombreux outils de création de maillages, de résolution de systèmes linéaires, ainsi que ses bibliothèques d'éléments finis, *etc.*

Nous introduisons les nouvelles routines d'optimisation désormais accessibles depuis la bibliothèque de modules du logiciel. En particulier, le logiciel libre d'optimisation sous contraintes IPOPT, qui implémente une méthode de points intérieurs très robuste pour l'optimisation en grande dimension. Nous appliquons avec succès ces algorithmes à une série de problèmes concrets parmi lesquels la résolution numérique de problèmes de surfaces minimales, la simulation de condensats de Bose-Einstein, ou encore un problème de positionnement inverse en mécanique des fluides.

Une version prototypique de FreeFem++ contenant les outils de différentiation automatique est présentée, après avoir exposé les principes fondamentaux de cette méthode de calcul de dérivées pour le calcul scientifique.

Mots-clefs

Calcul scientifique, développement de logiciel, différentiation automatique, méthode des éléments finis, équations aux dérivées partielles, optimisation numérique.

Optimisation tools for FreeFem++

Abstract

The goal of this Ph.D. thesis was the development of tools for the FreeFem++ software in order to make optimization problems easier to deal with. This has been accomplished following two main directions. Firstly, a set of optimization softwares is interfaced and validated before making use of them. Then, we analyse the field of automatic differentiation as a potential mean of simplification for the users.

FreeFem++ is an integrated development environment dedicated to numerically solving partial differential equations. Its high level language allows the user for a comfortable experience while using its mesh generation capabilities, linear system solvers, as well as finite elements capabilities, *etc.*

We describe the newly available optimization features, with a certain emphasis on the open source software IPOPT, which implements a state of the art interior points method for large scale optimization. These optimization tools are then used in a set of quite successful applications, among which minimal surfaces, Bose-Einstein condensate simulation, and an inverse positioning problem in the context of computational fluid dynamics.

Finally, after an introduction to the techniques of algorithmic differentiation, we also present an unstable prototype version of FreeFem++ including automatic differentiation features.

Keywords

Automatic differentiation, finite elements method, numerical optimization, partial differential equations, scientific computing, software development.

Table des matières

Introduction

L'optimisation dans FreeFem++

Quand il est question de réduction des coûts dans le domaine du calcul scientifique, le réflexe est de penser en terme d'amélioration des méthodes de calcul utilisées ou à un déploiement de puissance de calcul toujours plus grand. Ces deux approches ont bien sûr été, et restent probablement les plus fécondes. On pensait que la limite physique imposée par la taille des transistors mettrait un terme à la pérennité de la loi de Moore. Pourtant, avec la multiplication des cœurs à l'intérieur même des processeurs, celle-ci a trouvé une issue heureuse à son destin en apparence funeste, et la puissance des ordinateurs continue d'augmenter inlassablement. Les méthodes de calcul intensif et de *big data* participent à rendre l'utilisation de ces outils de calcul toujours plus astucieuse et productive, et les scientifiques ne cessent de mettre au point des méthodes numériques toujours plus efficaces.

Il est cependant une voie à laquelle on ne pense pas forcément et qui peut pourtant avoir un impact considérable sur la vitesse à laquelle on peut compter obtenir des résultats. Il s'agit du temps consacré à la phase de développement. Or, ce terme dissimule de nombreuses étapes, laborieuses et techniques. Car le développement, ce n'est pas uniquement le plaisir de mettre au point des programmes. C'est également celui de les tester, les déboguer, éprouver leurs performances et valider leur fonctionnement. Toutes ces phases, très coûteuses en temps, sont évidemment inévitables lors de la création de nouveaux outils. Mais on peut aussi souvent en faire l'économie. C'est dans cette perspective qu'a été créé FreeFem++

FreeFem++ est un logiciel destiné à la résolution numérique d'équations aux dérivées partielles, posées en dimension deux ou trois d'espace, par la méthode des éléments finis. Il se distingue de simples bibliothèques d'éléments finis en proposant un langage dédié qui montre une nette similitude avec l'écriture mathématique des problèmes pour lesquels le logiciel affirme son utilité, tout en restant proche des langages C et C++ : le mathématicien comme le programmeur y trouvent donc leur compte puisque le premier pourra rapidement mener ses expérimentations numériques sans avoir à trop se salir les mains, tandis que le second ne sera pas dérouté par l'apprentissage d'un langage complètement nouveau. Des outils de génération et d'adaptation de maillage facilitent la définition des problèmes aussi bien en dimension deux que dans l'espace. Cette thèse s'inscrit dans le cadre d'un important travail de développement de modules qui a été amorcé ces dernières années afin d'offrir des interfaces à d'autres logiciels ou bibliothèques qui sont alors directement accessibles via le script. Ainsi, FreeFem++ réunit dans une syntaxe commune et des interfaces à peu près standardisées, une vaste quantité d'outils susceptibles d'être exploités dans le cadre de la résolution numérique de problèmes par éléments finis, ce qui permet à l'usager de faire abstraction des phases de programmation les plus techniques et d'ainsi pouvoir se concentrer sur les aspects numériques et mathématiques du travail.

C'est dans cette optique de réduction des contingences de l'écriture des codes que s'ins-

crit cette thèse, en ciblant tout particulièrement le domaine de l'optimisation. Avant nos travaux, les solutions proposées par le logiciel dans ce domaine étaient très modestes : avec seulement deux algorithmes d'optimisation libre disponibles, la question de l'optimisation sous contraintes ne pouvait être abordée sans envisager l'écriture d'un programme dédié ou l'utilisation d'une routine d'optimisation externe, ce qui dans les deux cas impliquait une phase de développement relativement lourde. Le but de cette thèse a donc été de fournir à l'usager un éventail de solutions suffisamment étendu pour répondre à la plupart des situations rencontrées en optimisation numérique.

Pour ce type de problème, il existe deux grandes classes de méthodes de recherche de minimiseurs. Les algorithmes dits *classiques* ou à gradient qui tentent de réaliser certaines conditions nécessaires d'optimalité basées sur les dérivées de la fonction coût et des éventuelles contraintes, et exploitent avantageusement l'information spécifique apportée par les dérivées de ces fonctions. L'alternative à ces méthodes est d'employer les méthodes dites *heuristiques* ou *métaheuristiques* telles que les *algorithmes génétiques* ou *évolutionnistes*, ou encore *stochastiques* (recuit simulé, *etc.*). En dépit de leur capacité à déterminer le minimum global et de l'avantage considérable de ne pas supposer de régularité, ces méthodes sont fortement pénalisées par des vitesses de convergence relativement faibles et leur utilisation nécessitera donc d'importants temps de calcul, même avec de gros moyens.

Optimisation dérivable

Dans le cas d'un algorithme utilisant les informations fournies par les dérivées des fonctions définissant le problème, chaque itération est divisée en deux principales phases : la recherche d'une direction de descente $d_k \in \mathbb{R}^n$, suivie de la détermination d'un pas de descente ρ_k. L'itérée suivante est alors $x_{k+1} = x_k + \rho_k d_k$. Ces algorithmes se différencient alors par la manière dont ces deux étapes sont effectuées. De par la méthode employée pour calculer la direction de descente, on distingue par ailleurs trois grandes classes pour de tels algorithmes :

- les algorithmes à gradients dans lesquels $d_k = -\nabla f(x_k)$
- les algorithmes basés sur la méthode de Newton pour lesquels la direction de descente est la solution du système $\nabla^2 f(x_k) d_k = \nabla f(x_k)$
- les algorithmes de type *quasi*-Newton, dans lesquels on construit une approximation H_k de la matrice hessienne évaluée en les itérés, la direction étant alors, comme pour la méthode de Newton, la solution du système linéaire $H_k d_k = \nabla f(x_k)$

Il s'agit là des directions de descente pour des problèmes non contraints. Pour les problèmes avec contraintes d'égalité, on pourra introduire des multiplicateurs de Lagrange, mais il faudra alors modifier la méthode pour prendre en compte ces nouvelles variables. Le cas des contraintes d'égalité va nécessiter une phase supplémentaire d'analyse pour chaque itérée afin de déterminer quelles sont les contraintes effectivement actives qui vont engendrer un multiplicateur. L'alternative aux multiplicateurs de Lagrange est de se ramener à un problème non contraint, en remplaçant f par une fonction dite "de mérite" Φ dont on espère que, si elle est bien choisie, la minimisation aboutisse sur un minimiseur qui est aussi celui du problème d'origine. Une bonne fonction de mérite doit être la plus proche possible de f à l'intérieur du domaine délimité par les contraintes, et "grande" en dehors de celui-ci, tout en préservant la régularité des fonctions. Il est parfois nécessaire d'itérer sur la minimisation de plusieurs fonctions de mérite successives, de plus en plus proches de la fonction coût initiale. On pourra par exemple pénaliser f par une norme arbitraire mesurant la violation des contraintes : $\Phi(x) = f(x) + \frac{1}{\epsilon}\|c^+(x)\|_p^p$. Nous exposerons succinctement dans la section 2.4 une version améliorée de cette technique appelée *lagrangien augmenté*.

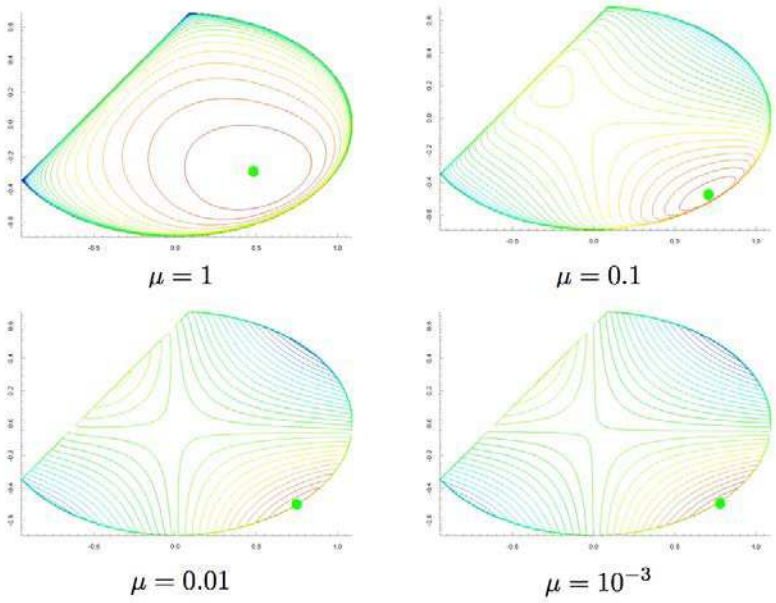


FIGURE 1: Fonction de mérite logarithmique avec différentes valeurs de μ pour la minimisation de $(x_1, x_2) \mapsto \frac{10}{3}x_1x_2 + \frac{1}{6}x_1$ sous les contraintes $\frac{19}{16} - x_1^2 - \frac{5}{2}x_2^2 \geq 0$ et $x_1 - x_2 + \frac{3}{5} \geq 0$ (exemple tiré de [44])

Lorsqu'il n'est pas possible de donner un sens à f pour des valeurs de x qui ne valident pas les contraintes, il est courant d'utiliser une fonction de mérite du type $\Phi(x, \mu) = f(x) + \mu I(c(x))$ où I préserve les propriétés de dérivabilité de c et f, et $I(s) \to +\infty$ lorsque $s \to 0^-$. C'est l'approche utilisée dans les *"barrier methods"*, et, par extension, les méthodes de *points intérieurs*, dans lesquelles on minimise $x \mapsto \Phi(x, \mu)$ pour des valeurs de μ de plus en plus petites (voir figures 1). A l'origine, c'est la fonction inverse qui était utilisée pour jouer le rôle de fonction barrière : $\Phi(x, \mu) = f(x) - \mu \sum_{i=1}^{m} 1/c_i(x)$. On lui préfère aujourd'hui le logarithme : $\Phi(x, \mu) = f(x) - \mu \sum_{i=1}^{m} \ln |c_i(x)|$, ce qui permet d'obtenir une approximation des multiplicateurs de Lagrange (voir 2.5 ou [44]). La figure 1 montre le graphe de $\Phi(x, \mu)$ pour une fonction coût, deux contraintes arbitraires, et différentes valeurs de μ. On y voit le minimiseur évoluer dans l'intérieur du domaine où les contraintes sont vérifiées et s'approcher de la frontière et de la solution réelle du problème.

Le problème de cette stratégie est de souvent mener à la résolution de systèmes linéaires mal conditionnés dans les cas où le minimiseur est proche de la frontière $c(x) = 0$. Certaines méthodes de points intérieurs raffinent le procédé et apportent une solution à ces difficultés comme nous le verrons dans la section 2.5.

Algorithmes évolutionnistes

Une démarche fondamentalement différente est celle des algorithmes d'optimisation dits *évolutionnistes* dans lesquels un ensemble fini d'éléments du domaine de définition de f, appelé *population*, est modifié à chaque itération par des méthodes inspirées des sciences du vivant. Dans les premières versions de ces algorithmes, l'évolution de la population est gouvernée par une série de *croisements* et de *mutations*, à définir selon le problème. Le croisement d'un certain nombre d'éléments de la population peut par exemple se faire par le choix aléatoire d'un élément de leur enveloppe convexe, quand la mutation pourra se faire par une petite perturbation aléatoire selon une ou plusieurs directions. A chaque

itération, un certain nombre de nouveaux points sont générés par ce procédé aléatoire puis ajoutés à la population, de laquelle on exclura les éléments dont l'image par la fonction coût est la plus grande. Certains algorithmes pratiquent cette dernière sélection de façon aléatoire avec une probabilité d'exclusion qui croît avec l'image par la fonction coût.

Si ces algorithmes offrent le gros avantage de parfois permettre la recherche de minima globaux et de pouvoir être parallélisés de manière triviale, leur vitesse de convergence est le plus souvent très inférieure à celle des algorithmes utilisant les dérivées. De plus, ils nécessitent un très grand nombre d'évaluations de la fonction coût. Certaines méthodes proposent des variantes visant à réduire la part de l'aléatoire et à exploiter plus d'information propre à la fonction, ce qui aboutit à des améliorations sensibles des propriétés de convergence dans certaines situations.

Une stratégie qui peut se révéler intéressante pour la minimisation de fonctions dérivables présentant de nombreux bassins d'attraction, mais localement convexes dans ceux-ci, est d'initialiser un algorithme à gradient (ou de type Newton) par le résultat d'un algorithme évolutionniste grossier (c'est-à-dire avec une population de taille relativement réduite ou peu d'itérations). Cette stratégie permet en quelque sorte de *"globaliser"* le résultat de la recherche d'un algorithme local.

Le développement d'outils d'optimisation de ce type était initialement prévu comme l'un des objectifs de cette thèse. L'efficacité discutable de ces méthodes, même lorsque les calculs sont massivement parallélisés, nous a cependant poussé à reconsidérer la pertinence d'un effort dans cette direction. Nous avons donc décidé d'abandonner le sujet après avoir pourvu FreeFem++ du minimum vital en la matière (voir section 2.6).

Nouveaux algorithmes pour FreeFem++

Dans la version 1.22 de FreeFem (mai 2002), trois algorithmes d'optimisation dérivable de la bibliothèque COOOL [64] ont été directement intégrés au noyau du logiciel. Il s'agit de méthodes bien connues et dont l'utilisation est largement répandue. On trouvera les détails techniques de ces algorithmes dans l'annexe A. Ces implémentations, en plus de ne pouvoir s'appliquer qu'à des problèmes sans contraintes et, pour la méthode de Newton, de requérir une implémentation des dérivées secondes, utilisent des matrices pleines de dimensions égales au nombre de paramètres d'optimisation. Leur utilisation est donc limitée à des problèmes de taille modeste.

On aura donc remarqué l'absence de méthodes naturellement dédiées au traitement de problèmes d'optimisation avec contrainte. Une limite assez drastique sur la dimension de l'espace des paramètres d'optimisation était également imposée à l'utilisateur, pour qui l'unique possibilité de contourner ces restrictions était l'écriture d'un algorithme adapté à son problème dans le script FreeFem++, ou éventuellement dans un module C++. Nous avons donc, au cours de cette thèse, œuvré pour élargir les possibilités offertes par le logiciel.

Dans cette perspective, nous avons implémenté des interfaces pour les algorithmes de la bibliothèque d'optimisation NLopt qui regroupe une grande quantité de méthodes d'optimisation allant des classiques tels que la méthode de Newton et BFGS, jusqu'à des méthodes stochastiques un peu ésotériques, en passant par des variantes de NEWUOA destinées à l'optimisation sous contraintes. On consultera le site des développeurs [60] pour une liste exhaustive de ces méthodes. L'intérêt majeur de cette bibliothèque est de pouvoir injecter très simplement un lagrangien augmenté dans chacun de ses algorithmes, ce qui constitue un choix assez vaste de solutions pour l'optimisation sous contraintes. Cette bibliothèque n'apporte toutefois pas de réelle solution pour l'optimisation en dimension élevée. Pour cet ultime problème, nous avons finalement travaillé sur l'interfaçage

du logiciel open source IPOPT. Cette interface pour IPOPT nous a par la suite permis de mener toute une série d'expérimentations numériques. L'une d'entre elles a débouché sur la publication d'un article dans la revue *Mathematical Modelling and Numerical Analysis*. Cet article a été inclus dans ce manuscrit au chapitre 3, section 3.5. IPOPT s'est finalement imposé comme l'outil de choix dans la *quasi*-totalité des applications que nous avons traitées, reléguant NLopt au rang de reliquat d'une époque révolue. Ces applications seront par ailleurs présentées au chapitre 3.

Différentiation automatique

Une grande majorité des algorithmes d'optimisation efficaces nécessite le calcul du gradient, ou même de dérivées secondes de la fonction coût. Dans un contexte numérique, la résolution d'un problème d'optimisation mènera donc souvent à répondre de manière adaptée à la question de l'évaluation de ces dérivées, qui n'est pas en général une question facile. L'utilisation de modèles discrétisés peut en effet introduire des phénomènes numériques inédits dans la version continue. Ainsi, s'il est possible par l'analyse au niveau continu, d'écrire et de résoudre numériquement un problème pour le calcul des dérivées, il se peut que celles-ci ne soient pas exactement celles du problème discrétisé initial, avec tout ce que cela peut impliquer de préjudiciable lors de l'utilisation d'un algorithme d'optimisation en aval. L'écriture des dérivées au niveau continu peut de plus être parfois relativement difficile, comme c'est notamment le cas en optimisation de forme lorsque la fonction coût est définie par une intégrale sur un domaine par rapport auquel la fonction est différentiée. L'alternative des techniques de différences finies, qui approchent effectivement les bonnes quantités, comporte quant à elle des difficultés intrinsèques auxquelles il est difficile d'échapper et ne peut donc être identifiée comme un candidat sérieux pour la conception d'un outil générique de différentiation.

Il existe néanmoins un ensemble de techniques qui calculent exactement, à la précision machine près, les dérivées de la fonction implémentée par un programme informatique par rapport à ses arguments. Il s'agit des méthodes de *différentiation automatique*, que nous présentons dans le chapitre 4. Ces techniques, qui trouvent leur fondement dans le principe élémentaire de dérivation des fonctions composées, nous ont semblé aller de concert avec la logique de simplification des codes prônée par la philosophie de FreeFem++. Le calcul automatisé des dérivées déchargerait complètement l'utilisateur du calcul de celles-ci, simplifiant à l'extrême l'utilisation des algorithmes d'optimisation pour lesquels il ne serait nécessaire que d'implémenter la fonction coût. Nous avons donc travaillé à l'introduction dans FreeFem++ d'une telle méthode de calcul des dérivées basée sur l'un des modes de *différentiation automatique*. Si ce travail s'est finalement révélé problématique et n'a abouti que sur une version du logiciel trop instable pour être distribuée, il nous a apporté une bonne connaissance de l'architecture du noyau de FreeFem++ qui aura été très précieuse lors du développement des outils d'optimisation.

Plan de thèse

Cette thèse aura donc été consacrée au développement d'outils d'optimisation pour FreeFem++, que ce soit par une implémentation directe d'interfaces pour des algorithmes, ou de fonctionnalités pour le calcul des dérivées, ainsi qu'à leur application à la résolution de problèmes numériques concrets. Ce manuscrit débutera donc par un chapitre de présentation du logiciel FreeFem++. Nous y exposons dans leurs grandes lignes les possibilités qu'offre le logiciel à travers quelques exemples plus ou moins classiques d'utilisation

et de tutoriel. Nous y décrirons également les quelques fonctionnalités avancées que nous utiliserons systématiquement dans les applications.

Le second chapitre décrit les algorithmes d'optimisation sur lesquels nous avons travaillé et que nous avons introduits dans FreeFem++. Cette partie commencera par le rappel des éléments théoriques nécessaires à la compréhension des méthodes d'optimisation qui y sont introduites. Nous exposerons ensuite une série d'applications de ces algorithmes à des problèmes concrets d'optimisation que nous avons traités. On commencera par quelques problèmes de surfaces minimales qui auront essentiellement servi de test de validation. La seconde application concerne un problème d'optimisation de position en mécanique des fluides, traité dans le cadre du *Data-Base Workshop for multiphysics optimization software validation*. En troisième section de ce chapitre, nous présentons une simulation de condensat de Bose-Einstein par minimisation de l'énergie de Gross-Pitaevsky. Enfin, nous incluons un article co-écrit avec Zakaria Belhachmi, Faker Ben Belgacem et Frédéric Hecht dans lequel il sera question d'un problème de contact avec conditions aux bords de Signorini, pour lequel notre contribution aura consisté à fournir les données utilisées pour illustrer numériquement les propos théoriques.

Le dernier chapitre traitera de la différentiation automatique, d'abord dans ses principes puis en pratique. Nous présenterons aussi le logiciel que nous avons baptisé FreeFem++-ad, notre prototype FreeFem++ dans lequel nous avons introduit la différentiation automatique. Si les trois premiers chapitres gagnent à être lus dans l'ordre où ils apparaissent, celui-ci est relativement indépendant et pourra être abordé à tout moment.

Chapitre 1

Le logiciel `FreeFem++`

Cette partie est consacrée à l'utilisation de FreeFem++ au travers d'exemples et de brèves descriptions des fonctionnalités auxquelles nous avons été le plus souvent confrontés au cours de cette thèse. Il n'est pas question de produire ici une documentation alternative complète, mais plutôt de donner une description d'ensemble des capacités du logiciel, tout en précisant certaines astuces que nous avons copieusement utilisées et qui, même si certaines peuvent apparaître furtivement dans la documentation officielle [79], ne font pas systématiquement l'objet d'explications toujours satisfaisantes et étaient, pour nombre d'entre elles, plutôt énigmatiques.

1.1 Brève histoire de FreeFem++

FreeFem++ a été initié par la commercialisation en 1987 par Olivier Pironneau de ses deux ancêtres, MacFEM et PCFEM, écrits en Pascal et dédiés à la résolution d'équations aux dérivées partielles en 2D. En 1992, le code est retranscrit en C++ et le logiciel bascule dans le domaine libre en devenant FreeFem, incluant le générateur de maillage bamg et des éléments finis P1 et P2. Le logiciel sera par la suite enrichi d'une algèbre de fonctions en 1996 pour devenir FreeFem+, puis FreeFem++ suite à un remaniement complet lors duquel le langage utilisateur muni de la syntaxe qu'on lui connaît aujourd'hui a été développé. Une première tentative d'extension à la dimension trois sera proposée en 1999 sous l'impulsion d'Olivier Pironneau et développée par S. Del Pino, en se basant sur une méthode de domaines fictifs, mais il faudra attendre la seconde réécriture du noyau éléments finis de 2008 pour que le développement de modules de calcul 3D réellement fonctionnels soit initié dans la troisième version de FreeFem++.

Aujourd'hui FreeFem++ bénéficie d'un succès certain dans le domaine académique et la recherche, avec une communauté de plus de deux mille utilisateurs et une liste de diffusion très active. Les domaines de prédilection de FreeFem++ sont avant tout la recherche académique et l'enseignement, car le logiciel permet de rapidement implémenter et tester de nouveaux algorithmes, ou de réaliser diverses expérimentations numériques avec une relative aisance. Il offre un environnement idéal pour le mathématicien désirant éprouver numériquement une idée en s'affranchissant des contraintes des langages de programmation usuels. Il s'inscrit donc dans la continuité des logiciels du type MatLab ou SciLab, toute proportion gardée, avec la particularité de se spécialiser dans la résolution d'EDP par éléments finis. Le logiciel demande un minimum de connaissances quant aux mathématiques qui sous-tendent la résolution des problèmes. Aussi, son utilisation dans le cadre d'applications industrielles est-elle peut-être moins répandue. La syntaxe, pour laquelle la notion de formulation faible (communément appelée variationnelle) joue un rôle central,

pourra en effet souvent dérouter physiciens et ingénieurs, plus coutumiers des formulations fortes des équations. Le logiciel a cependant démontré ses capacités pour ce genre d'utilisation avec la résolution effective d'un problème à plus d'une dizaine de milliards d'inconnues par Pierre Jolivet en 2013 sur le super-calculateur Curie du CEA.

1.2 FreeFem++ et son langage

FreeFem++ met à disposition de l'utilisateur un langage impératif fortement typé regroupant certains éléments syntaxiques du C et du C++, ainsi que certaines fonctionnalités de haut niveau qui lui sont propres et qui permettent de réaliser des tâches complexes avec facilité. Ce langage est exploité par l'intermédiaire d'un script dont les commandes sont interprétées puis exécutées par le programme principal du logiciel. En tant que langage interprété, les opérations de bas niveau les plus élémentaires sont nécessairement réalisées avec une efficacité modeste au regard de la vitesse à la quelle celles-ci pourraient l'être par un programme compilé dédié. Aussi, l'intérêt de FreeFem++ est de regrouper sous une syntaxe commune relativement épurée les fonctions de haut niveau d'algèbre linéaire et de résolution par éléments finis qui, quant à elles, sont effectivement exécutées par le programme compilé écrit en C++, et donc à vitesse optimale. L'avantage sur l'écriture de tâches équivalentes dans un langage de programmation compilé réside dans la simplification extrême du code, en permettant d'abstraire l'utilisateur de techniques d'implémentation fastidieuses, mais aussi dans le fait de manipuler une syntaxe présentant une certaine similitude avec l'écriture des problèmes mathématiques pour lesquels le logiciel affirme son utilité.

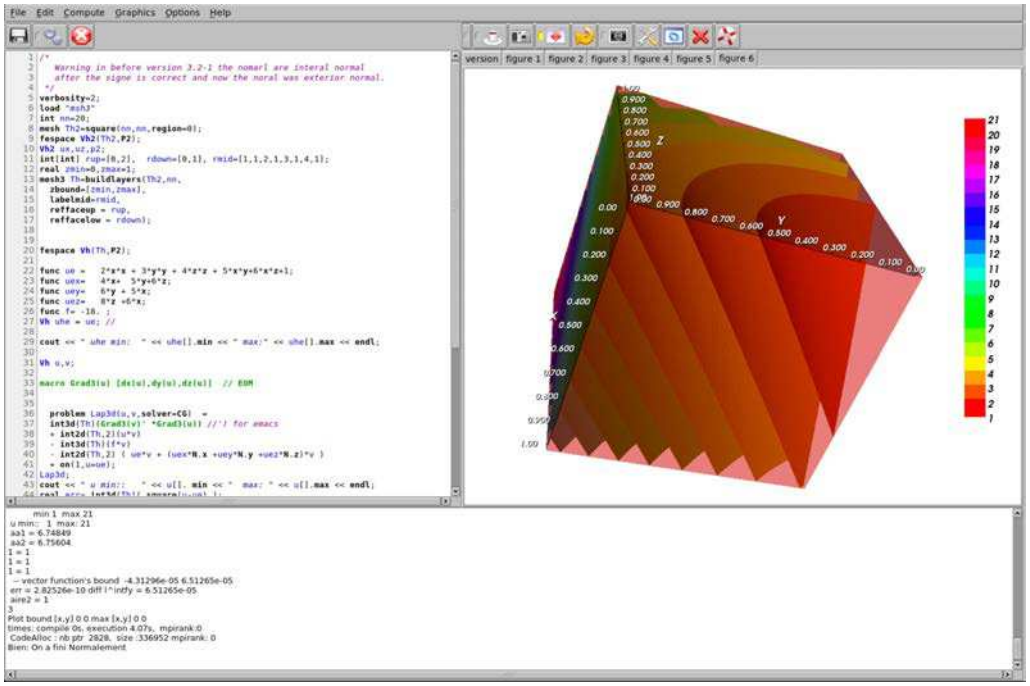


FIGURE 1.1: FreeFem++-cs : l'environnement de développement intégré pour FreeFem++

On utilise FreeFem++ par l'intermédiaire d'un script, l'appel au logiciel se faisant le plus souvent en ligne de commande. Il existe aussi une version pourvue d'une interface graphique appelée *FreeFem++-cs*, développée et maintenue par Antoine Le Hyaric. Elle regroupe un éditeur de script avec coloration syntaxique, une fenêtre de visualisation, ainsi

qu'une zone spécifique aux informations relatives à la compilation et à l'exécution.

Le choix qui a été fait pour ce qui est du style syntaxique est de s'orienter fortement vers celui du C et du C++, en utilisant les accolades { et } pour délimiter les blocs syntaxiques, ainsi que le point virgule pour délimiter les instructions. La syntaxe pour les boucles `for` et `while` est également la même que pour les langages précédemment cités, de même que pour les blocs conditionnels `if`/`else`. Comme dans la plupart des autres langages, l'utilisateur a la possibilité de déclarer et définir des variables des types numériques usuels : `double` et `long int` qui correspondent respectivement aux types `real` et `int` de FreeFem++, mais aussi des variables booléennes `bool`, des nombres complexes `complex` et des chaînes de caractères `string`. Ces variables peuvent être manipulées par le biais des opérateurs usuels du C++, dont la plupart trouvent leur équivalent dans FreeFem++ (affectation =, opérateurs arithmétiques +, -, *, /, opérateurs hybrides conjuguant une opération arithmétique à l'affectation +=, -= *etc.*, opérateurs de comparaison ==, !=, %, *etc.*, opérateurs booléens &&, ||, *etc.*). Bien entendu, tous les opérateurs définis dans les langages C et C++ ne sont pas pertinents dans le contexte d'un logiciel tel que FreeFem++. C'est pourquoi seul un échantillon des plus utiles a fait l'objet d'une transposition vers cette syntaxe. On consultera la documentation pour la liste complète de ces opérateurs.

Les tableaux, dont les données sont de l'un des types numériques `int`, `real` ou `complex`, peuvent être introduits avec les déclarateurs `int[int]`, `real[int]` *etc.* Ils peuvent être manipulés avec les opérations vectorielles les plus simples :

```
real[int] a=...,b=...; //déclare et définit les vecteurs a et b
real lambda = 4;
real[int] c = a + b; //déclare et définit c, somme des vecteurs a et b
real[int] d = -a, e = a-b, f = lambda*a; //autres opérations
real ps = a'*b; //produit scalaire
```

Des tableaux à deux indices, considérés comme matrices pleines peuvent également être déclarés et manipulés, le déclarateur correspondant étant `real[int,int]` (ou `int[int,int]` pour une matrice à coefficients entiers).

Ces tableaux ressemblent plus au `valarray` de la bibliothèque standard du C++ qu'à de simples vecteurs, puisqu'il est possible d'appliquer une fonction à l'ensemble de leurs éléments par un simple appel, mais surtout car, comme pour le `valarray`, les opérations les plus courantes ont fait l'objet d'une optimisation évitant les recopies inutiles causées par les mécanismes internes de passage d'arguments et de renvoi d'objets du C++. Pour ces raisons techniques, lorsqu'un calcul implique des tableaux dans un script FreeFem++, seules les expressions les plus simples y sont autorisées :

```
real[int] g = a + lambda*b; //ok
real[int] h = a + b + c; //erreur de compilation
```

L'erreur provoquée par la définition de `h` est typique des restrictions induites par cette optimisation, seules les combinaisons linéaires a deux vecteurs ayant été implémentées.

On distingue dans FreeFem++ trois types principaux de fonctions selon leur mécanisme de déclaration. Tout d'abord, la possibilité de définir les fonctions de manière similaire à ce que l'on pourrait faire en C/C++, avec la différence que la définition doit immédiatement suivre la déclaration :

```
func real[int] UneFonction(real a,int n,real[int] &array)
{
        //définition de la fonction
```

```
        return AnotherArray //renvoie un tableau
}
```

Plus spécifique à FreeFem++, la possibilité de définir des fonctions dépendant implicitement des variables spatiales par l'intermédiaire de leurs identifiants réservés `x`, `y` et `z`. Le typage de ces fonctions est laissé à la discrétion du compilateur et est donc omis dans le script :

```
func f = cos(x) + cos(y);                    func g = x + 1i*y;
```

Le troisième type de fonction est bien sûr celui des fonctions éléments finis que nous aborderons dans une section spécifique.

Enfin, un préprocesseur permet l'inclusion d'autres fichiers contenant du script FreeFem++ à l'aide de la commande `include`, mais aussi la déclaration de macro-définitions réalisant la substitution automatique de chaînes de caractères, avec ou sans arguments.

1.3 Génération de maillages

L'un des atouts de FreeFem++ réside dans sa capacité à générer très simplement des maillages pour des problèmes en deux dimensions, et dans les différents outils permettant la manipulation de ces maillages. Par l'intermédiaire d'une interface simple, on appelle des fonctions réalisant des tâches relativement complexes, sans avoir à utiliser de logiciel spécifiquement dédié à la création et à la modification des triangulations en dehors de FreeFem++, ce qui permet de travailler efficacement sur les liens qu'entretiennent les solutions des simulations numériques avec le maillage sur lequel elles sont calculées, que ce soit en dimension deux ou trois.

1.3.1 Triangulations de Delaunay

L'outil de base pour la création de maillages dans FreeFem++ est la commande `buildmesh` qui, à partir d'une liste de fonctions définissant le contour orienté, réalise une triangulation de Delaunay du domaine que l'on souhaite mailler. Si la frontière Γ du domaine est par exemple décrite par un jeu de n fonctions $F_i : [a_i, b_i] \to \mathbb{R}^2$, de sorte que $\Gamma = \bigcup_{i=1}^{n} \Gamma_i$ avec $\Gamma_i = \{F(t)\ ,\ t \in [a_i, b_i]\}$, avec l'orientation $s_i \in \{+, -\}$, le script FreeFem++ permettant la génération d'une triangulation à partir d'une telle frontière dont chacun des Γ_i serait discrétisé en m_i points est le suivant :

```
1 border Gamma1(t=a1,b1) {x=F1x(t); y=F1y(t); label=1;}
2 border Gamma2(t=a2,b2) {x=F2x(t); y=F2y(t); label=2;}
3 ...
4 border Gamman(t=an,bn) {x=Fnx(t); y=Fny(t); label=n;}
5
6 mesh Th = buildmesh(Gamma1(±m1)+Gamma2(±m2)+...+Gamman(±mn));
```

Le signe + spécifiant une orientation positive pour un arc de courbe pourra être omis.

L'appel de `buildmesh` en ligne 6 va générer une triangulation de Delaunay à partir des points de l'ensemble $\bigcup_{i=1}^{n} \{F_i(a_i + k(b_i - a_i)/m_i\ ,\ 0 \leq k \leq m_i\}$. Il est aussi possible de générer une grille uniforme sur le carré unité à partir de la commande `square`, à laquelle on pourra éventuellement appliquer une transformation. Deux maillages peuvent également être assemblés lorsque la correspondance entre les points de la frontière commune est parfaite. Un simple signe + permet d'effectuer ce collage. Le maillage 1.8 illustre

l'exploitation de toutes ces fonctionnalités. Il est bien entendu tout à fait possible d'utiliser des maillages préalablement créés en dehors de FreeFem++, pourvu que ceux-ci soient sauvegardés dans le format adéquat (se référer à la documentation [79]).

```
int np=5;//finesse du maillage
int[int] lab=[1,5,3,4];//étiquetage des bords

mesh Th1 =
  square(2*np,2*np,[2*(x-1),2*(y-0.5)],label=lab);

border b1(t=-pi/2.,pi/2.)
  {x=cos(t); y=sin(t); label=2;}
border b2(t=0,1)
  {x=0.; y=1-2*t; label=5;}
mesh Th2 = buildmesh(b1(pi*np) + b2(2*np));
mesh Th = Th1 + Th2;

plot(Th,wait=1);
```

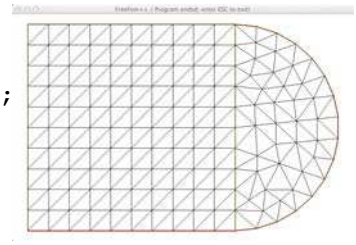


Figure 1.2: Maillage généré par le script ci-contre

On trouvera ensuite une panoplie d'outils importants pour l'édition de maillages. `movemesh` permet par exemple d'appliquer une transformation à l'ensemble des points du maillage, en conservant la connectivité de celui-ci.

```
//inversion de centre (-3,0), puis rotation
//d'angle pi/3
//en écriture complexe

complex Z0 = -3,Z1=exp(1i*pi/3);

//Affixe complexe du point courant :
func Z = x + (1i)*y;

func X = real(Z1*(Z0 + 1./(Z-Z0)));
func Y = imag(Z1*(Z0 + 1./(Z-Z0)));

Th = movemesh(Th,[X,Y]);

plot(Th,wait=1);
```

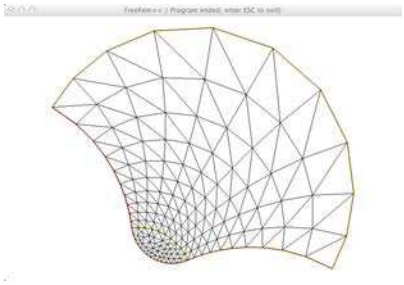


Figure 1.3: Maillage déformé par inversion et rotation.

Les routines `splitmesh` et `trunc` s'utilisent pour subdiviser les éléments d'une triangulation. La première en découpant les triangles en fonction d'une application dont la valeur en le centre de chaque simplex donne le nombre d'éléments en lequel celui-ci doit être découpé. Quant à `trunc`, elle prend en argument, en plus du maillage initial, un niveau de fragmentation, et une fonction indicatrice qui va permettre d'identifier quels triangles seront morcelés. Ces deux méthodes fournissent un premier ensemble rudimentaire de solutions au problème de l'adaptation de maillage, que nous couvrirons plus en détail dans une section dédiée, l'outil de choix pour cette tâche étant la routine `adaptmesh` que nous décrirons, et qui adapte astucieusement le maillage aux variations d'une fonction donnée.

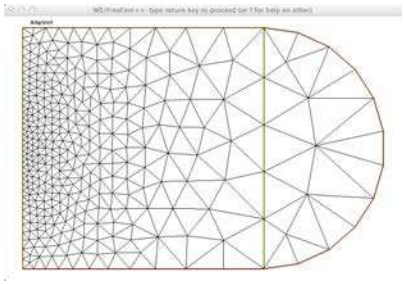
Au-delà de l'économie de degrés de liberté dans les zones de variations des solutions, l'adaptation de maillage est aussi utile pour la conception de maillage lorsque l'on utilise des transformations de l'espace. Ces transformations n'étant que rarement des isométries,

elles ont tendance à resserrer ou au contraire à espacer les points des maillages auxquels elles sont appliquées, et il n'est absolument pas certain que cette nouvelle répartition des points soit conforme aux variations des fonctions impliquées dans les calculs pour lesquels le maillage en question est conçu. La métrique du système de coordonnées curvilignes défini par la transformation du plan (ou de l'espace) $T = (X, Y)$ appliquée au maillage se calcule comme le produit de la transposée de la jacobienne de cette transformation J par elle-même :

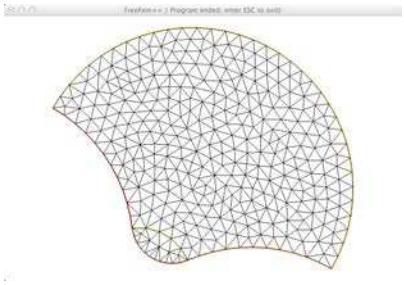
$$M = J^{\mathrm{T}} J = \begin{bmatrix} \left(\frac{\partial X}{\partial x}\right)^2 + \left(\frac{\partial Y}{\partial x}\right)^2 & \frac{\partial X}{\partial x}\frac{\partial X}{\partial y} + \frac{\partial Y}{\partial x}\frac{\partial Y}{\partial y} \\ \frac{\partial X}{\partial x}\frac{\partial X}{\partial y} + \frac{\partial Y}{\partial x}\frac{\partial Y}{\partial y} & \left(\frac{\partial X}{\partial y}\right)^2 + \left(\frac{\partial Y}{\partial y}\right)^2 \end{bmatrix} \tag{1.1}$$

L'adaptation du maillage par cette métrique, avant application de la transformation, permet d'obtenir une distribution de points uniforme dans la triangulation finale. Nous nous proposons de retourner à l'exemple 1.8/1.3 pour illustrer cet astucieux procédé, en reprenant le code après création du premier maillage en ligne 16 :

```
16 real h = 1./np;
17
18 //Transformation :
19 complex Z0 = -3,Z1=exp(1i*pi/3);
20 func Z = x + (1i)*y;
21 func X = real(Z1*(Z0 + 1./(Z-Z0)));
22 func Y = imag(Z1*(Z0 + 1./(Z-Z0)));
23
24 //Coefficients de la jacobienne
25 func Xx = real(-Z1/square(Z-Z0));
26 func Xy = real(-1i*Z1/square(Z-Z0));
27 func Yx = imag(-Z1/square(Z-Z0));
28 func Yy = imag(-1i*Z1/square(Z-Z0));
29 func m11 = Xx^2 + Yx^2;
30 func m12 = Xx*Xy + Yx*Yy;
31 func m22 = Xy^2 + Yy^2;
32 real v=1./h^3;
33
34 Th = adaptmesh //on adapte...
35      (Th,m11*v,m12*v,m22*v,IsMetric=true);
36 plot(Th,wait=1,cmm="Adaptated");
37 Th = movemesh(Th,[Tx,Ty]); //...et on déforme.
38
39 plot(Th,wait=1);
```



(a) Maillage adapté à la transformation



(b) maillage final

1.3.2 Les maillages en dimension trois

Concevoir des maillages de l'espace est une opération bien plus délicate qu'en dimension 2 dès que l'on a affaire à des géométries un tant soit peu compliquées. FreeFem++ permet l'utilisation et la manipulation de maillages tétraédriques, par la commande `readmesh3` pour le chargement de géométries préalablement existantes, ou en utilisant les différents outils de conception de maillages pour en créer un à la volée. Pour cette tâche, on pourra

par exemple utiliser l'interface FreeFem++ de TetGen [90], qui permet de construire un maillage tétraédrique à partir de n'importe quel domaine 3D polyédrique. Les surfaces délimitant le domaine à mailler peuvent être construites par transformation de maillages de dimension deux à l'aide de la commande `movemesh23`. Le maillage de la boule unité, de laquelle a été extraite une plus petite boule pourra par exemple se faire comme suit :

```
load "msh3"
load "tetgen"
load "medit"

int np=10;                          //densité de points sur la frontière
real xc=0.3,yc=0.,zc=0.;            //centre de la sphère interne
real rc=0.4;                        //rayon de la sphère interne
real h1=1./np,h2=1./(rc*np);        //longueur des arêtes
real vol = 0.0025;
mesh Th = square(2*pi*np,pi*np,[2*pi*x,pi*y-pi/2.]);

func X = cos(y)*cos(x); //Transformation définissant la sphère
func Y = cos(y)*sin(x);
func Z = sin(y);
func Xx = -cos(y)*sin(x);           //Jacobienne pour le calcul
func Xy = -sin(y)*cos(x);           //de la métrique
func Yx = cos(y)*cos(x);
func Yy = -sin(y)*sin(x);
func Zx = 0.;
func Zy = cos(y);
func M11 = Xx^2  + Yx^2  + Zx^2;          //Métrique
func M12 = Xx*Xy + Yx*Yy + Zx*Zy;
func M22 = Xy^2  + Yy^2  + Zy^2;
real v1 = 1/h1^2 , v2=1/h2^2;
func periodicity = [ [2,y] , [4,y] ];

mesh Th1 =
   adaptmesh(Th,M11*v1,M12*v1,M22*v1,IsMetric=1,periodic=periodicity);
mesh Th2 =
   adaptmesh(Th,M11*v2,M12*v2,M22*v2,IsMetric=1,periodic=periodicity);
mesh3 Ext =
   movemesh23(Th1,transfo=[X,Y,Z]) +
   movemesh23(Th2,transfo=[xc+rc*X,yc+rc*Y,zc+rc*Z]);

real[int] reg = [0.,0.,0.,1.,vol], hl=[xc,yc,zc];
//Tetraedrisation :
mesh3 Ball =
   tetg(Ext,switch="paAAQY",regionlist=ref,nbofholes=1,holelist=hl);

medit("Ball",Ball);//pour visualiser avec médit
```

Adapter le maillage 2D à la métrique définie par la jacobienne de la transformation permet d'obtenir une triangulation uniforme sur les surfaces 3D. Dans certains cas, comme celui de notre exemple, il se peut même que la phase de tetraédrisation du volume délimité par la surface échoue à cause de certains points singuliers. La régularisation apportée par

l'adaptation du maillage à la transformation permet d'éviter ce type de problèmes tout en augmentant la qualité du maillage final (voir l'exemple de la figure 1.5). Au regard de la longueur de ce script, alors que l'on ne construit qu'une géométrie relativement simple, on comprend que le maillage d'objets complexes se révélera être une tâche laborieuse pour laquelle FreeFem++ n'est peut-être pas encore bien adapté. Sans doute vaut-il mieux construire ces maillages en utilisant un logiciel dédié, puis l'importer à l'aide de `readmesh3`.

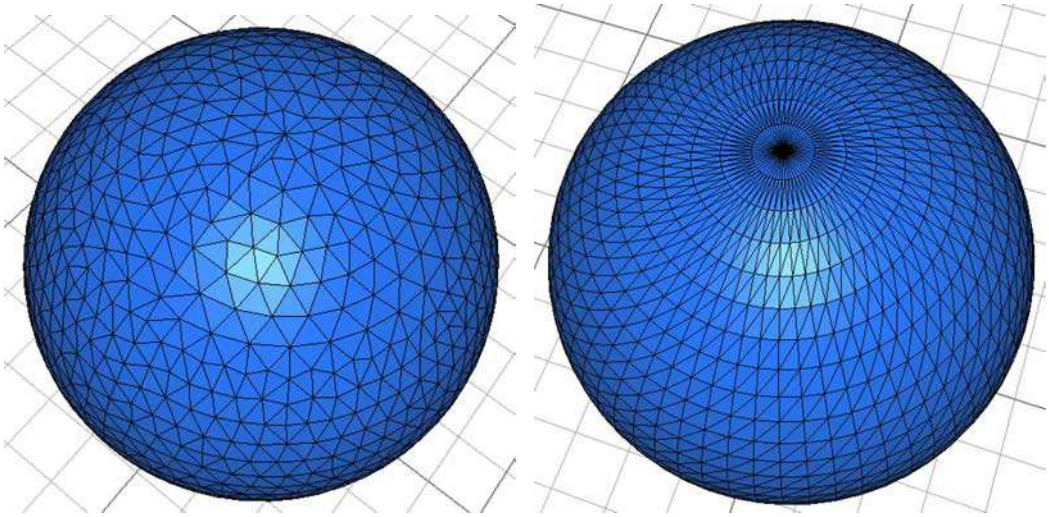


FIGURE 1.5: Triangulation d'une sphère avec et sans adaptation du maillage 2D.

L'utilisateur bénéficie également d'outils tels que `tetgconvexhull` qui construit une tétraédrisation de Delaunay de l'enveloppe convexe d'un ensemble de points de l'espace. Mentionnons `buildlayermesh` qui permet, à partir de deux fonctions $z_{\min}$ et $z_{\max}$ de $\mathbb{R}^2$ dans $\mathbb{R}$ telles que $z_{\min} < z_{\max}$, de donner une dimension supplémentaire à une triangulation d'un domaine Ω_{2D} du plan en maillant avec des tétraèdres la région de l'espace :

$$\Omega_{3D} = \left\{(x, y, z) \in \mathbb{R}^3 \mid (x, y) \in \Omega_{2D} \text{ et } z_{\min}(x, y) \leq z \leq z_{\max}(x, y)\right\}$$

Des exemples d'utilisations de ces fonctions pourront bien évidemment être trouvés dans la documentation du logiciel.

1.4 Adaptation de maillages

Il est fréquent que les solutions calculées dans le cadre de simulations numériques comportent de brusques variations sur certaines régions du domaine de calcul tout en présentant des zones de relative platitude. Il est évident qu'un maillage trop grossier ne permet pas d'obtenir une description précise d'une fonction présentant de telles variations. Inversement, un maillage uniformément fin fournira sur la fonction des informations redondantes, source de calculs superflus. Dès lors il est naturel d'envisager de travailler avec des maillages *adaptés* à la solution, avec des mailles plus grossières dans les régions où la solution varie peu, tout en générant plus de degrés de liberté dans les régions où leur accumulation est pertinente.

L'un des points forts de FreeFem++ est de proposer des outils efficaces permettant une telle adaptation au travers d'une interface simple. Nous nous intéressons dans cette section à la méthode d'adaptation de maillages utilisée dans FreeFem++. Sans entrer trop

en profondeur dans les détails de la génération de triangulations de type Delaunay, nous décrirons les critères qui permettent d'obtenir un maillage bien adapté à une fonction par analyse *a posteriori*.

Nous aborderons également la question essentielle de l'adaptation de maillage pour les systèmes d'équations, dans lesquels toutes les composantes du vecteur solution ne présentent pas nécessairement un profil de variation similaire aux autres et dont chacune d'elles peut prendre des valeurs avec des ordres de grandeur sans commune mesure de l'une à l'autre. Nous nous baserons pour cela essentiellement sur les travaux [71], [17], [22], [23] et [72]. La question de l'adaptation de maillage pour des problèmes instationnaires ne sera pas évoquée et nous renvoyons le lecteur aux références proposées dans le très synthétique [72] et notamment aux travaux de F. Alauzet.

1.4.1 Quelques arguments théoriques

Le problème de l'adaptation optimale de maillage d'un domaine borné et polygonal Ω de $\mathbb{R}^2$ à une fonction f est défini comme la minimisation de l'erreur d'approximation parmi toutes les triangulations de cardinalité donnée. D'un point de vue théorique, il s'agit d'un problème difficile d'approximation dite *non-linéaire* donnant lieu à de nombreuses questions ouvertes. Compte tenu de ces difficultés théoriques, l'adaptation de maillage en pratique s'est pendant longtemps limitée à découper ou regrouper récursivement les éléments du maillage respectivement trop grands ou trop petits, jusqu'à satisfaction d'un indicateur d'erreur, souvent basé sur le gradient de f. Cependant, dans les cas où les variations de la fonction par rapport à laquelle l'adaptation est réalisée présentent de fortes anisotropies, cette méthode n'est pas optimale, puisqu'elle débouche sur une accumulation de nombreux degrés de liberté dans les zones de grandes variations en scindant les éléments de manière complètement isotrope.

Plus récemment, des méthodes d'adaptation, anisotropes par essence, se basant sur des métriques construites à partir des dérivées secondes de la fonction à approcher ont été introduites. D'une efficacité certaine, ces méthodes trouvaient leur justification dans des arguments heuristiques et ne disposaient pas d'un cadre théorique solide, jusqu'à ce que des travaux relativement récents [26] démontrent rigoureusement leur efficacité.

Les auteurs montrent qu'on peut construire une suite $(\mathcal{T}_n)_{n\geq n_0}$ de triangulations de Ω telles que $\forall n$, $\mathcal{T}_n$ comporte au plus n éléments, et telle que :

$$\limsup_{n\to\infty} n \left\| f - \Pi_{\mathcal{T}_n} f \right\|_{L^p(\Omega)} \leq C \left\| \sqrt{|\det(\nabla^2 f)|} \right\|_{L^\tau(\Omega)} \tag{1.2}$$

où τ vérifie $\frac{1}{\tau} = 1 + \frac{1}{p}$ et C est une constante indépendante de p, Ω et f. Au reste, cette borne se trouve être optimale pour une classe de triangulations dont la caractérisation est naturellement vérifiée en pratique.

Pour établir l'estimation d'erreur 1.2, les auteurs construisent une suite de triangulations dont les attributs métriques et les orientations des mailles sont localement gouvernés respectivement par les valeurs propres et les vecteurs propres de $\nabla^2 f$. Pour cela, on définit une métrique riemannienne M par :

$$M = \frac{1}{h^2} \left(\det \nabla^2 f \right)^{-\frac{1}{2p+2}} \nabla^2 f \tag{1.3}$$

dans laquelle h est une constante permettant de contrôler la finesse globale de la triangulation. A tout point z de Ω, M associe une matrice $M(z)$ définie positive, qui caractérise

une ellipse :

$$\mathcal{E}_z = \left\{ x \in \mathbb{R}^2 \mid \ x^{\mathrm{T}} M(z) x \leq 1 \right\}$$

On dit alors qu'une triangulation $\mathcal{T}$ est adaptée à M si, pour tout $T \in \mathcal{T}$, il existe deux constantes α_1 et α_2 telles que $0 < \alpha_1 < \alpha_2$, et :

$$b_T + \alpha_1 \mathcal{E}_{b_T} \subset T \subset b_T + \alpha_2 \mathcal{E}_{b_T} \tag{1.4}$$

b_T désignant le barycentre de T. En d'autres termes, une triangulation est adaptée à la métrique M si tous ses triangles sont *presque* équilatéraux dans cette métrique. On montre alors qu'une telle triangulation vérifie :

$$n \left\| f - \Pi_{\mathcal{T}} f \right\|_{L^p(\Omega)} \leq C \left\| \sqrt{|\det(\nabla^2 f)|} \right\|_{L^\tau(\Omega)} \tag{1.5}$$

où C dépend de α_1 et α_2. On se rapproche donc de l'estimation d'erreur optimale 1.2. Ces idées sont généralisées au cas de l'interpolation d'ordre quelconque dans [71], ce qui devrait très certainement aboutir dans un futur proche sur le développement de nouveaux outils pour l'adaptation de maillages à des fonctions éléments finis de Lagrange d'ordre élevé.

1.4.2 Les outils `adaptmesh` et `mmg3d`

Les outils d'adaptation de maillage de FreeFem++ `bamg` et `mmg3d` permettent, à partir d'une métrique donnée, de concevoir un maillage adapté à cette métrique dans le sens 1.4. En dimension 2, la métrique est automatiquement calculée à partir de la fonction que l'on passe à la routine d'adaptation qui se chargera d'en calculer les dérivées secondes. `mmg3d` ne travaille qu'avec une métrique, aussi sera-t-il nécessaire d'utiliser la routine de calcul de métrique de `mshmet` préalablement. Si cette hessienne se diagonalise comme :

$$\nabla^2 f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x^2} & \dfrac{\partial^2 f}{\partial x \partial y} \\ \dfrac{\partial^2 f}{\partial y \partial x} & \dfrac{\partial^2 f}{\partial y^2} \end{bmatrix} = R \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} R^{-1} \tag{1.6}$$

alors, en se donnant une longueur maximale $h_{\max}$ et une longueur minimale $h_{\min}$ pour les arêtes du nouveau maillage, ainsi qu'un indicateur de précision générale ϵ, la métrique utilisée est la suivante :

$$\mathcal{M} = \frac{1}{\epsilon} R \begin{bmatrix} \tilde{\lambda}_1 & 0 \\ 0 & \tilde{\lambda}_2 \end{bmatrix} R^{-1} \tag{1.7}$$

dans laquelle :

$$\tilde{\lambda}_i = \min \left(\max \left(|\lambda_i| , \frac{1}{h_{\max}^2} \right) \ , \ \frac{1}{h_{\min}^2} \right) \tag{1.8}$$

Lorsque le maillage doit épouser simultanément les variations de deux fonctions, f_1 et f_2, la métrique $\mathcal{M}$ utilisée résulte d'une opération dite d'*intersection* des métriques $\mathcal{M}_1$ et $\mathcal{M}_2$ que l'on obtiendrait par le procédé précédemment décrit pour l'adaptation à chacune de ces fonctions prises séparément. Cette intersection s'obtient par moyennage de deux métriques intermédiaires $\hat{\mathcal{M}}_1$ et $\hat{\mathcal{M}}_2$ telles que, si on note respectivement $R_1 = [r_1^1 \ r_2^1]$ et

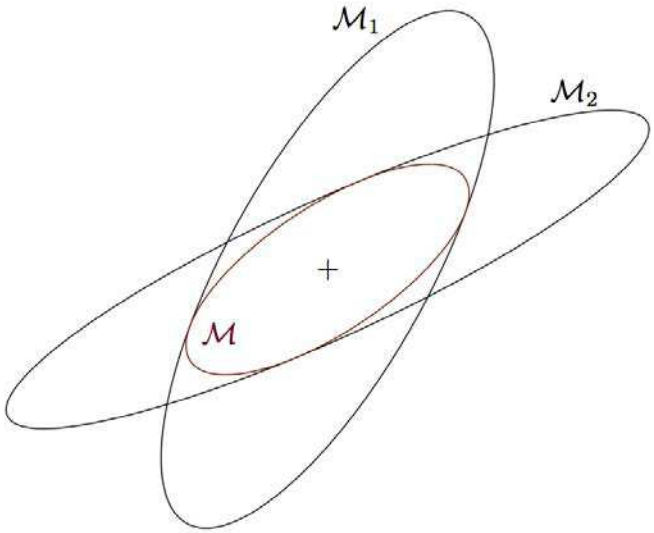


Figure 1.6: Intersection de deux métriques en 2D

$R_2 = [r_1^2\ r_2^2]$ les matrices des vecteurs propres de $\mathcal{M}_1$ et $\mathcal{M}_2$, et λ_i^j les valeurs propres de $\mathcal{M}_j$:

$$\mathcal{M} = \frac{\hat{\mathcal{M}}_1 + \hat{\mathcal{M}}_2}{2} \quad \text{où} \quad \hat{\mathcal{M}}_j = R_j \begin{bmatrix} \hat{\lambda}_1^j & 0 \\ 0 & \hat{\lambda}_2^j \end{bmatrix} R_j^{-1}$$
$$\text{avec} \begin{cases} \hat{\lambda}_i^1 = \max\left(\lambda_i^1, {r_i^1}^{\mathrm{T}} \mathcal{M}_2 r_i^1\right) \\ \hat{\lambda}_i^2 = \max\left(\lambda_i^2, {r_i^2}^{\mathrm{T}} \mathcal{M}_1 r_i^2\right) \end{cases} \tag{1.9}$$

Géométriquement, en tout point $z \in \Omega$, la métrique $\mathcal{M}$ réalisant l'intersection est celle qui définit une ellipse $\mathcal{E}_z$ dont les axes principaux sont de longueurs maximales et vérifie $\mathcal{E}_z \subset \mathcal{E}_z^1 \cap \mathcal{E}_z^2$ où $\mathcal{E}_z^1$ et $\mathcal{E}_z^2$ sont les ellipses respectivement engendrées par $\mathcal{M}_1$ et $\mathcal{M}_2$ en z (voir figure 1.6).

En pratique, adapter un maillage peut nécessiter plusieurs tentatives successives, notamment quand le maillage de départ est très grossier. Partant d'une triangulation initiale $\mathcal{T}_0$, et en se donnant un nombre d'itérations N_{adapt}, on pourra par exemple itérer selon le schéma très simple suivant :

for $k = 0$ **to** N_{adapt} **do**
  $u_k \leftarrow$ solution de (E) posé sur $\mathcal{T}_k$
  $\mathcal{T}_{k+1} \leftarrow \text{adaptmesh}(\mathcal{T}_k, u_k)$
end for

Dans ce court algorithme, (E) peut aussi bien désigner un procédé de résolution d'équation par éléments finis que l'interpolation d'une fonction sur un espace de fonctions éléments finis.

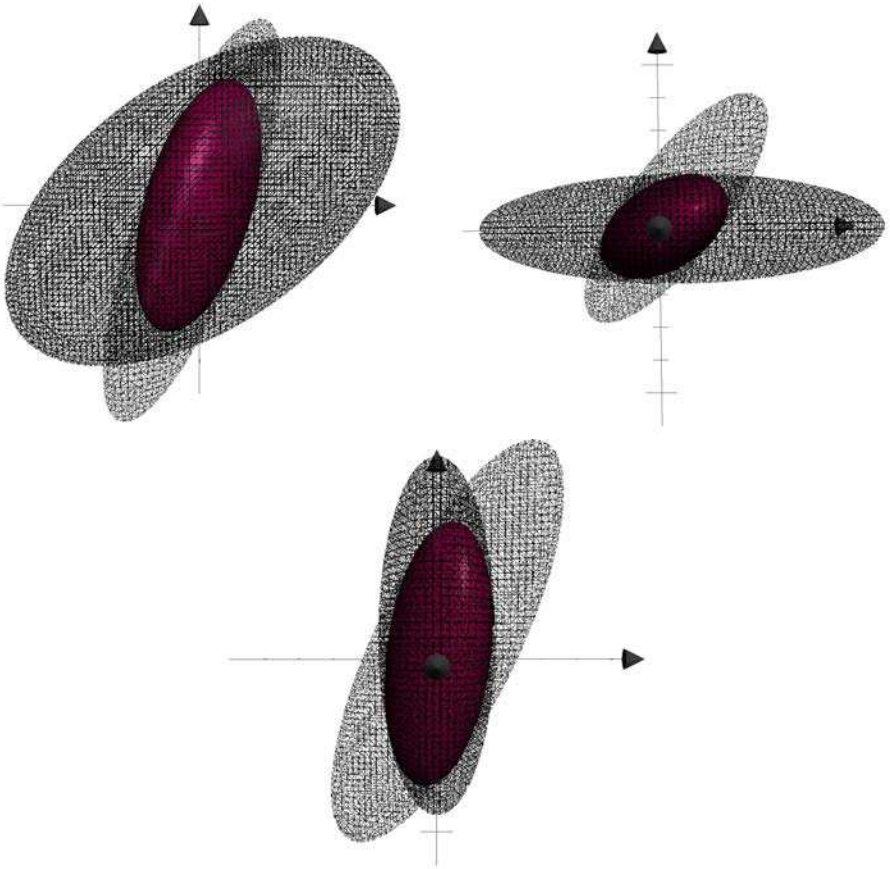


FIGURE 1.7: Intersection de deux métriques en dimension 3 - vue depuis différentes directions

1.5 Eléments finis

1.5.1 Définition d'espaces de fonctions éléments finis

Le mot clé `fespace` permet de déclarer un espace de fonctions éléments finis en associant un maillage à un type d'éléments finis. Le type d'éléments est à choisir parmi une liste d'identifiants prédéfinis dans le langage. L'utilisateur ne peut *a priori* pas définir ses propres types d'éléments finis dans un script, l'adjonction d'une telle fonctionnalité nécessitant l'écriture d'un module en C++. Les éléments finis directement accessibles dans le logiciel sont tout d'abord les éléments de Lagrange d'ordre 0, 1 et 2, que l'on nomme le plus souvent par les abréviations P^0, P^1 et P^2. Pour une triangulation donnée, les fonctions de l'espace d'éléments finis P^k sont les applications dont chaque restriction à l'un des éléments de la triangulation est une fonction polynomiale de degré k. Des éléments de Lagrange d'ordres plus élevés sont accessibles dans la bibliothèque de modules où l'on trouvera des fichiers pour la prise en compte d'éléments d'ordres 3 et 4.

Au-delà de ces éléments les plus couramment utilisés, certains types d'éléments plus exotiques sont également proposés à l'utilisateur. On trouvera par exemple l'opportunité d'utiliser des éléments P1 dits *non conformes*, P1-*bulle*, ainsi que les éléments de Raviart-Thomas, *etc.* La documentation du logiciel dresse une liste complète, accompagnée de brèves descriptions synthétiques de ces types d'éléments finis.

```
1 mesh Th = ... ; //Un maillage
2
```

```
//Déclaration d'espaces de fonctions éléments finis
fespace Vh(Th,P1); //fonctions scalaires
fespace V2h(Th,[P1,P1]); //fonctions vectorielles
fespace Wh(Th,[P2,P2,P1]);
```

L'opérateur d'affectation = joue ici le rôle d'opérateur de projection pour la définition de fonctions éléments finis lorsque celles-ci sont initialisées par une fonction des deux variables x et y ou encore une fonction d'un autre type d'éléments finis. Pour chaque espace de fonctions éléments finis, il est possible de déclarer des fonctions à valeurs dans le corps des nombres complexes :

```
//Définition de fonctions éléments finis
Vh uh=sin(2*pi*x)*cos(pi*x);
V2h [ux,uy]=[x,y];
Wh [u1,u2,p]=[ux,uy,uh];
Vh<complex> zh = ux + (1i)*uy;
```

1.5.2 Matrices d'interpolation

Si l'opérateur d'affectation "=" appelé sur une fonction éléments finis réalise automatiquement l'interpolation , il peut parfois être utile de disposer de la matrice de l'application linéaire définie par l'interpolation d'un espace de fonctions éléments finis vers un autre. Une commande a donc été prévue à cette fin :

```
matrix M = interpolate(Vh2,Vh1,... /*paramètres optionnels*/...);
```

Munie de cette matrice, la manipulation suivante explicite en quelque sorte ce qui se passe lorsque l'on écrit `Vh2 u2=u1;`, à cela près que dans ce dernier cas, la matrice n'est pas assemblée :

```
Vh1 u1 = ... ;//Définition d'une FEF de Vh1
Vh2 u2;//Simple déclaration
u2[] = M*u1[];
```

Les paramètres optionnels permettent de sélectionner un opérateur différentiel spécifique, par exemple lorsque l'on désire construire la matrice d'interpolation entre les dérivées par rapport à x des fonctions éléments finis, ou encore une composante dans le cas de fonctions vectorielles. Le code suivant procède par exemple à l'extraction de la première et de la troisième composante :

```
fespace Wh(Th,[P2,P2,P2,P1]);
fespace Vh(Th,[P2,P2]);
int[int] u2vc = [1,3];
matrix M = interpolate(Vh,Wh,U2Vc=u2vc);

Wh [ux,uy,uz,p] = ... ; //déclaration et définition
Vh [Ux,Uz];//déclaration
Ux[] = M*ux[]; // équivalent à [Ux,Uz]=[ux,uz];
```

Cette fonctionnalité nous sera en particulier utile pour assembler de manière élégante les matrices jacobiennes et hessiennes des fonctions intervenant dans certains problèmes d'optimisation afin de les passer aux routines de minimisation.

1.6 De l'art de résoudre des problèmes variationnels

Revenons à présent vers le but premier de FreeFem++ : la résolution d'équations aux dérivées partielles par la méthode des éléments finis. Et c'est dans ce domaine que le logiciel dispose de l'avantageuse particularité de permettre à l'utilisateur de définir ces problèmes par leur formulation variationnelle abstraite. L'exemple du problème de Poisson, posé sur un ouvert $\Omega \subset \mathbb{R}^2$ illustre éloquemment la similitude des deux écritures.

1.6.1 L'équation de Poisson

Ce problème bien connu consiste à, étant donnée une fonction f continue par morceaux sur Ω, trouver une fonction u, *a priori* de classe C^2, telle que $-\Delta u = f$ dans Ω et $u = 0$ sur $\partial\Omega$. L'écriture variationnelle de ce problème permet de relâcher la contrainte assez forte sur la régularité de u. Elle s'obtient par intégration par partie de l'équation après l'avoir multipliée par une fonction test à support compact dans Ω, et suppose au second membre d'être de carré intégrable. On recherche alors u dans l'espace de Sobolev $H_0^1(\Omega)$ vérifiant :

$$\forall v \in H_0^1(\Omega) \ , \ \int_\Omega \nabla u.\nabla v - \int_\Omega fv = 0 \tag{1.10}$$

Le passage au numérique se fait tout d'abord par le choix d'une triangulation $\mathcal{T}_h$ de Ω, h désignant le diamètre maximal des triangles (ou tétraèdres pour le cas 3D) de la triangulation, puis par celui d'un type d'éléments finis, ce qui revient à choisir un espace vectoriel sur $\mathbb{R}$ de dimension finie V_h tel que $V_h \subset H_0^1(\Omega)$, compte tenu du problème que l'on cherche à résoudre. Si on note φ_i, $1 \leq i \leq N$ les fonctions de base de V_h (avec $N = \dim(V_h)$), le problème 1.10 posé dans V_h devient : trouver $u_h \in V_h$, telle que :

$$\forall i \in [\![1, N]\!] \ , \ \int_{\mathcal{T}_h} \nabla u_h.\nabla \varphi_i - \int_{\mathcal{T}_h} f\varphi_i = 0 \tag{1.11}$$

En tant qu'élément de V_h, u_h se développe sur la base de fonctions éléments finis en $u_h = \sum_{j=1}^N u_j\varphi_j$. La résolution de 1.11 consiste donc à déterminer les composantes u_j, $1 \leq j \leq N$ de u_h. L'injection de ce développement dans 1.11 aboutit au système linéaire suivant :

$$\forall i \in [\![1, N]\!] \ , \ \sum_{j=1}^N u_j \int_{\mathcal{T}_h} \nabla\varphi_i.\nabla\varphi_j = \int_{\mathcal{T}_h} f\varphi_i \tag{1.12}$$

En notant $U = (u_i)_{1\leq i\leq N}$ et $B = \left(\int_{\mathcal{T}_h} f\varphi_i\right)_{1\leq i\leq N}$, éléments de $\mathbb{R}^N$, ainsi que $A = (a_{ij})_{1\leq i,j\leq N}$ la matrice carrée d'ordre N de coefficients $a_{ij} = \int_{\mathcal{T}_h} \nabla\varphi_i.\nabla\varphi_j$, et en omettant les conditions au bord, 1.11 prend une écriture matricielle très compacte : $AU = B$. La résolution numérique de ce problème d'équations aux dérivées partielles est donc à présent un simple problème de résolution de système linéaire. De plus, les fonctions de base φ_i possèdent souvent un support relativement peu étendu, en principe restreint aux quelques éléments jouxtant le support du degré de liberté auquel elles sont associées, si bien qu'il existe peu de couples $(i, j) \in [\![1, N]\!]^2$ tels que $\text{supp}(\varphi_i) \cap \text{supp}(\varphi_j) \neq \emptyset$. Ainsi, la matrice A est de grande dimension, mais la plupart de ses coefficients sont nuls. Une telle matrice est dite *creuse*, et la résolution de systèmes linéaires impliquant une matrice de ce type nécessite des techniques particulières.

Une question demeure, celle de la condition au bord $u = 0$ sur $\partial\Omega$. Son traitement dépend du type d'éléments finis. Dans le cas d'éléments P^1, les degrés de liberté de l'espace d'éléments finis coïncident avec les valeurs prises en chaque point de la triangulation. On construira donc l'ensemble $\Gamma = \{i \in [\![1, N]\!] \mid p_i \in \partial\Omega\}$, p_i désignant les points de la

triangulation $\mathcal{T}_h$. Pour tout indice i appartenant à Γ, la ligne correspondante dans 1.12 doit alors être remplacée par $u_i = 0$. La matrice et le second membre du système sont donc :

$$a_{ij} = \begin{cases} \delta_{ij} & \text{si } i \in \Gamma \\ \displaystyle\int_{\mathcal{T}_h} \nabla\varphi_i.\nabla\varphi_j & \text{si } i \notin \Gamma \end{cases} \qquad b_i = \begin{cases} 0 & \text{si } i \in \Gamma \\ \displaystyle\int_{\mathcal{T}_h} f\varphi_i & \text{si } i \notin \Gamma \end{cases} \tag{1.13}$$

L'un des soucis de cette méthode est qu'elle transforme un problème initialement symétrique défini positif en un problème non symétrique. Dans FreeFem++, on utilise une alternative exploitant le caractère fini de la représentation des nombres en informatique, baptisée *pénalisation exacte*, afin de conserver la symétrie du problème. Son principe est de choisir un réel M, appelé `tgv` dans le logiciel, suffisamment grand pour que tout nombre apparaissant dans les coefficients de la matrice ou dans les composantes du second membre, possède une mantisse disjointe de chacune de celles des autres après multiplication par M. En double précision, $M = 10^{30}$ est suffisant si les ordres de grandeur des coefficients de la matrice ne varient pas de manière fantaisiste (auquel cas il est fort probable que celle-ci soit dramatiquement mal conditionnée). Muni de ce très grand nombre, ainsi que de la matrice A avec tous ses coefficients comme dans 1.12, plutôt que d'annuler les lignes d'indices appartenant à Γ, on les conserve entières en imposant à l'élément diagonal a_{ii} la valeur M. Quant au second membre, on annule aussi les composantes d'indice dans Γ. Si la condition au bord est non homogène, avec $u = g$ sur $\partial\Omega$, il faut prendre $b_i = Mg_i,\ \forall i \in \Gamma$, où $(g_i)_{1\leq i\leq N}$ sont les composantes de la projection de g sur V_h :

$$a_{ij} = \begin{cases} M & \text{si } i \in \Gamma \text{ et } i = j \\ \displaystyle\int_{\mathcal{T}_h} \nabla\varphi_i.\nabla\varphi_j & \text{si } i \neq j \end{cases} \qquad b_i = \begin{cases} Mg_i & \text{si } i \in \Gamma \\ \displaystyle\int_{\mathcal{T}_h} f\varphi_i & \text{si } i \notin \Gamma \end{cases} \tag{1.14}$$

Avec de telles modifications, toute composante d'indice $i \in \Gamma$ d'un produit matrice-vecteur Ax réalisé par un ordinateur est exactement égale à Mx_i, avec pour effet miraculeux d'imposer les valeurs au bord aux composantes adéquates et une résolution en conséquence du reste du système. Dans un gradient conjugué, par exemple, la première itération imposera la valeur au bord aux composantes d'indice dans Γ, après quoi celles-ci ne seront plus jamais modifiées, de même que les composantes correspondantes dans le vecteur résidu seront toujours exactement nulles.

FreeFem++ a été conçu pour réaliser toutes ces étapes par la simple définition du problème. Nous avons vu dans les sections précédentes comment construire une triangulation. La définition du problème de Poisson se fait en utilisant l'un des mot-clés `solve`, `problem` ou `varf`, selon que l'on désire une résolution automatique immédiate, différée ou manuelle par construction explicite de la matrice et du second membre. En dimension deux, ce problème est résolu en quelques lignes :

```
1 mesh Th = square(30,30); //maillage
2 fespace Vh(Th,P1);
3 func f=(x-0.5)*(y-0.5);
4 Vh u,v; //inconnue / fonction test
5 solve Poisson(u,v) =
6     int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))        //  ∫_Th ∇u.∇v
7   - int2d(Th)(f*v)                              //- ∫_Th fv
8   + on(1,2,3,4,u=0);                            //  u = 0 sur ∂Ω
9 plot(u,wait=1,nbiso=60,dim=3);
```

Une résolution dans l'espace du même problème ne nécessiterait que de substituer les mots clés `mesh3` et `int3d` à `mesh` et `int3d`, respectivement, conjointement à l'adjonction du terme associé aux dérivées partielles par rapport à z dans la première intégrale :

```
6   int3d(Th)(dx(u)*dx(v)  + dy(u)*dy(v)
7                          + dz(u)*dz(v))
```

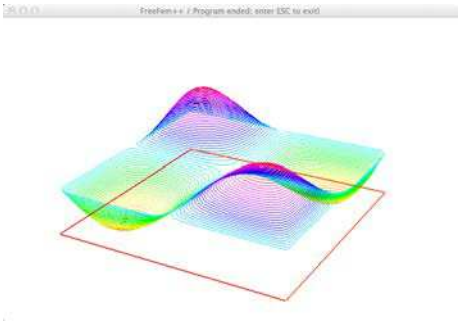


FIGURE 1.8: Solution de l'équation de Poisson avec $\Omega =]0,1[^2$ et $f = \left(x - \frac{1}{2}\right)\left(y - \frac{1}{2}\right)$

L'instruction `solve` assemble la matrice et le second membre comme explicité en 1.14 en imposant les conditions au bord par pénalisation exacte. L'analyseur syntaxique distingue automatiquement la partie bilinéaire, utilisée pour la construction de la matrice, de la forme linéaire nécessaire à celle du second membre du système. Selon la symétrie du problème et les options à l'installation, le logiciel détermine ensuite une routine de résolution de systèmes linéaires creux, en optant pour un gradient conjugué dans les cas, comme celui-ci, où la forme bilinéaire est symétrique, ou encore pour UMFPACK ou GMRES dans le cas contraire. Cette seule commande occulte donc l'exécution d'un lourd processus technique, permettant à l'utilisateur de s'affranchir de nombreux détails d'implémentation et de s'investir plus volontiers dans l'analyse et l'expérimentation mathématiques.

1.6.2 Cas général

Toutes les équations ne présentent pas le niveau extrême de simplicité de l'équation de Poisson, et il est bien entendu possible de résoudre numériquement des équations aux dérivées partielles plus complexes, qu'il s'agisse de problèmes linéaires ou non linéaires, ou encore de problèmes d'évolution, réels ou complexes. Dans tous les cas, on se ramène néanmoins à la résolution de problèmes linéaires sur le corps choisi. La description formelle la plus générale du problème élémentaire que peut résoudre FreeFem++ est la suivante : étant données une forme bilinéaire B et une forme linéaire L sur un espace de fonctions éléments finis V, trouver, si il existe, un élément $u \in V$ tel que :

$$\forall v \in V,\ B(u,v) - L(v) = 0$$

Le plus souvent ces deux formes, bilinéaire et linéaire, s'expriment par des intégrales sur une triangulation $\mathcal{T}_h$ d'un ouvert Ω de $\mathbb{R}^2$ (ou $\mathbb{R}^3$), si bien que l'on peut écrire le problème générique résolu par le logiciel sous la forme suivante :

$$\forall v \in V, \int_\Omega b\left((u,\nabla u),(v,\nabla v)\right) + \int_{\partial\Omega} \beta\left((u,\nabla u),(v,\nabla v)\right) - \int_\Omega l(v,\nabla v) - \int_{\partial\Omega} \lambda(v,\nabla v) = 0$$

où b et β sont bilinéaires, et l et λ linéaires. Le logiciel assemblera matrices et second membre :

$$\begin{aligned} a_{ij} &= \int_\Omega b\left((\varphi_i,\nabla\varphi_i),(\varphi_j,\nabla\varphi_j)\right) + \int_{\partial\Omega} \beta\left((\varphi_i,\nabla\varphi_i),(\varphi_j,\nabla\varphi_j)\right) \\ b_i &= \int_\Omega l(\varphi_i,\nabla\varphi_i) + \int_{\partial\Omega} \lambda(\varphi_i,\nabla\varphi_i) \end{aligned} \tag{1.15}$$

Dans ces intégrales, nous confondons l'ouvert Ω avec la triangulation $\mathcal{T}_h$. Il faut en réalité attribuer à $\int_\Omega$ le sens de $\sum_{T\in\mathcal{T}_h}\int_T$. Il est d'ailleurs possible avec FreeFem++ de définir un troisième type d'intégrale pouvant contribuer à la partie bilinéaire :

$$\sum_{T\in\mathcal{T}_h}\int_{\partial T}\alpha\left((u,\nabla u),(v,\nabla v)\right)$$

Les conditions aux bords sont imposées avec les modifications décrites en 1.14 ou par introduction de termes de bords issus de l'intégration par partie des équations pour les conditions de type Neumann.

L'étape ultime est la résolution du système linéaire obtenu $AU = B$. On dispose pour cela de plusieurs méthodes telles que le gradient conjugué lorsque la matrice est symétrique, définie positive, GMRES, ainsi que des méthodes directes de décomposition telles que UMFPACK, certaines pouvant être exécutées en parallèle dans la version MPI du logiciel. En résumé, les tâches essentielles réalisées par FreeFem++ consistent en un assemblage automatique de matrices et seconds membres dans le cadre des éléments finis, avec un puissant outil de projection de fonctions sur les espaces d'éléments finis, ainsi que divers outils de résolution de systèmes linéaires. Il s'agit là du minimum vital nécessaire à la résolution d'équations aux dérivées partielles.

1.6.3 Problèmes d'évolution

Pour résoudre un problème d'évolution, l'utilisateur doit avant tout se ramener à une succession de problèmes stationnaires, par exemple par le choix d'un schéma aux différences finies permettant la discrétisation des opérateurs différentiels en temps. On doit alors trouver, pour tout $n\in\mathbb{N}$, une fonction $u^n\in V$ telle que $\forall v\in V,\ B(u^n,v)-L(v)=0$, la forme linéaire L dépendant de certaines des solutions aux temps précédents u^{n-1}, parfois u^{n-2} et plus rarement des solutions plus en amont dans la résolution, selon le schéma de discrétisation et l'ordre des dérivées. Un problème d'évolution non linéaire nécessitera une étape supplémentaire pour le traitement de la non-linéarité, éventuellement par une méthode des caractéristiques, lorsque l'équation s'y prête.

La version instationnaire de l'équation de la chaleur, pour la résolution de laquelle quelques lignes de code suffisent dans FreeFem++, illustre bien le type de code nécessaire à la résolution de problèmes d'évolution. Dans sa forme classique, il s'agit de trouver une fonction de $C([0,T]\times\Omega)$, pour un certain $T>0$, vérifiant :

$$\begin{cases} \dfrac{\partial u}{\partial t}-\Delta u=0 & \text{dans }]0,T]\times\Omega \\ u=0 & \text{sur }]O,T]\times\partial\Omega \\ u(0,x)=u_0(x) & \forall x\in\Omega \end{cases} \tag{1.16}$$

La fonction u_0, constituant une donnée du problème, appartient à $L^2(\Omega)$. Afin d'affaiblir la notion de solution pour ce problème, la fonction u n'est pas recherchée dans un espace fonctionnel sur $[O,T]\times\Omega$, mais comme une fonction d'une seule variable à valeurs dans un espace fonctionnel X sur Ω du type $C^k([0,T],X)$. Cet espace varie selon les auteurs et le problème à résoudre, mais dans tout les cas, la continuité par rapport à la donnée étant une propriété essentielle, on pourra prendre (voir [4] ou [19]) :

$$u\in L^2\left(]0,T[,H_0^1(\Omega)\right)\cap C\left([0,T],L^2(\Omega)\right) \tag{1.17}$$

Une solution faible de 1.16 est alors un élément de 1.17 vérifiant :

$$\forall t \in]0,T[,\ \forall v \in H_0^1(\Omega)\ ,\ \frac{d}{dt}\int_\Omega u(t)v + \int_\Omega \nabla u(t).\nabla v = 0 \quad \text{et} \quad u(0) = u_0 \tag{1.18}$$

Une première discrétisation en espace, dans laquelle $H_0^1(\Omega)$ est approché par un espace de fonctions éléments finis V_h sur une triangulation $\mathcal{T}_h$ de Ω, de fonctions de bases φ_i, de façon similaire à ce qui est fait pour le problème stationnaire 1.10, aboutit à la résolution d'un grand système d'équations différentielles ordinaires (de dimension $N = \dim(V_h)$) sur les composantes U_h dans V_h de la solution :

$$\forall t \in]0,T[,\ M_h\frac{dU_h}{dt} + A_hU_h = 0$$

où $A_h = (\int_\Omega \nabla\varphi_i.\nabla\varphi_j)_{1\le i,j\le N}$ et $M_h = (\int_\Omega \varphi_i\varphi_j)$.

On achève la discrétisation totale du problème par le choix d'un schéma de différences finies pour la dérivée en temps. Nous arrêtons ici notre choix à celui d'un schéma implicite, inconditionnellement stable. Le problème revient alors à déterminer une suite de composantes $(U_h^n)_{n\in\mathbb{N}}$ définie par $U_0^h = \Pi_{V_h}(u_0)$, projection de la donnée sur V_0, et :

$$\forall n \in \mathbb{N}, M_h\frac{U_h^{n+1} - U_h^n}{\Delta t} + A_hU_h^{n+1} = 0 \tag{1.19}$$

Dans le script FreeFem++, on peut utiliser la commande `varf` pour définir et assembler les matrices de masse et de rigidité et écrire un script proche de l'équation complètement discrétisée ci-dessus 1.19, mais il est aussi possible, en utilisant `solve`, de se rapprocher de l'écriture 1.18, à la différence près que l'opérateur différentiel de temps doit être explicitement discrétisé puisque FreeFem++ ne le reconnaîtra pas. C'est ce que nous faisons dans le script suivant, que nous proposons pour la résolution de ce problème d'évolution.

```
1  int[int] labels=[1,1,1,1]; //inutile de distinguer les bords
2  mesh Th = square(10,10,label=labels);
3  fespace Vh(Th,P1);
4  real T = 2;//résolution pour t ∈ [0,2]
5  int n = 60;//précision de la discrétisation en temps
6  real dt=T/n;
7  func data = 16.*x*(1-x)*y*(1-y);
8  Vh unm1;
9  unm1 = data;
10 for(int i=1;i<n;++i)
11 {
12         Vh un,v;
13         solve Heat(un,v,init=i-1) = //pour ne pas reconstruire
14                     //et refactoriser la matrice à chaque itération
15             int2d(Th)(1./dt*un*v + dx(un)*dx(v) + dy(un)*dy(v))
16           - int2d(Th)(1./dt*unm1*v)
17           + on(1,un=0);              // u = 0 sur le bord
18         unm1=un;
19         plot(un,cmm="u at t="+(dt*i),value=1);
20 }
```

L'approche matricielle est cependant plus rapide, avec des temps de calcul plus proches de ceux que nécessiterait un programme dédié écrit en C. Il faut pour cela écrire explicitement le système linéaire dont chaque instantanée est la solution :

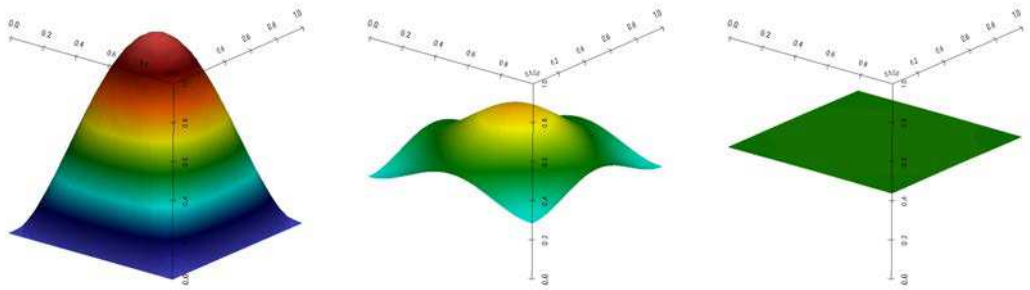
$$\forall n,\ (M_h + \Delta t A_h)\, U_h^{n+1} = M_h U_h^n$$

On utilisera `varf` pour directement construire les matrices $W_h = M_h + \Delta t A_h$ et M_h, cette dernière permettant de construire le second membre par un produit matrice-vecteur plutôt qu'en parcourant tous les éléments du maillage. L'ajout de l'option `solver=sparsesolver` dans la définition de W_h permet de factoriser la matrice à sa création, plutôt que de le faire à chaque itération. Le code suivant est l'équivalent optimisé du précédent. On considère que les déclarations de maillages et d'espaces éléments finis y sont identiques. On obtient, pour cet exemple, un facteur proche de trois lorsque `n` devient grand entre les temps d'exécution des deux programmes.

```
21 varf vfW(u,v) =
22     int2d(Th)(dt*dx(u)*dx(v) + dt*dy(u)*dy(v) + u*v)
23   + on(1,u=0);
24 varf vfM(u,v) = int2d(Th)(u*v);
25 varf vfOn1(u,v) = on(1,u=1);
26 matrix Wh = vfW(Vh,Vh,solver=sparsesolver);
27 matrix Mh = vfM(Vh,Vh);
28 real[int] on1 = vfOn1(0,Vh,tgv=1);
29
30 unm1 = data;
31 Vh un;
32
33 for(int i=1;i<n;++i)
34 {
35   real[int] rhs = B*unm1[];
36   rhs = on1 ? 0. : rhs;
37   un[] = Wh^-1 * rhs;
38   unm1 = un;
39   plot(un,cmm="u at t="+(dt*i),value=1);
40 }
```

On observe une décroissance du champ de température vers la valeur moyenne de la donnée initiale. Physiquement, ce problème modélise un espace clos complètement isolé dont la température initialement inhomogène s'harmonise avec le temps (figures 1.9).

Peut-être le traitement des problèmes d'évolution dans FreeFem++ pourrait-il faire l'objet de futurs développements. On pourrait imaginer une syntaxe plus spécifique, avec un opérateur de dérivation en temps `dt`, libérant l'utilisateur de la charge d'implémenter la boucle sur le temps et de celle de déclarer les objets nécessaires pour le stockage des instantanés requis pour l'assemblage des seconds membres, regroupant en quelques mot-clés les lignes de code de l'exemple ci-dessus. Toutefois,il s'agirait là essentiellement de masquer quelques détails d'implémentation pouvant s'avérer parfois redondants. Sans apporter de modifications cruciales ou ouvrir la voie vers la résolution de nouveaux types de problèmes avec FreeFem++, ce petit outil participerait à la simplification et à l'harmonisation de la syntaxe qui, dans ce cas, reste bien écartée de la formulation mathématique.



(a) $t = 0$ **(b)** $t > 0$ **(c)** $t \to +\infty$

FIGURE 1.9: Divers instantanés de l'équation de la chaleur homogène sur $[0,1]^2$ avec condition aux bords de Neumann homogènes et donnée initiale $u_0 = 16xy(1-x)(1-y)$, de valeur moyenne $\int_\Omega u_0 = \frac{4}{9}$.

1.6.4 Problèmes non linéaires

Tout comme les problèmes d'évolution, les formulations variationnelles non-linéaires ne peuvent être déclarées directement dans FreeFem++, aussi ce type de problème nécessite-t-il un traitement particulier. Il n'y a cependant pas ici de procédé systématique, ce qui fait que la résolution de ces problèmes est très contextuelle et souvent difficile du point de vue numérique.

Par exemple, si la solution au problème variationnel réalise le minimum d'une fonctionnelle, on pourra s'attaquer à la minimisation directe de celle-ci en invoquant l'une des routines d'optimisation interfacées dans FreeFem++. C'est le cas dans les problèmes de surfaces minimales, que nous aborderons dans la section 3.2, dans laquelle, plutôt que de chercher à résoudre la complexe équation exprimant la nullité de la courbure moyenne, on minimise directement la fonctionnelle d'aire.

Dans les cas où l'on est capable de trouver une bonne initialisation à celle-ci, on peut essayer d'appliquer la méthode de Newton. Nous décrirons celle-ci dans le chapitre consacré à l'optimisation, à la section 2.1. Formellement, si la formulation variationnelle non linéaire consiste à trouver u dans un certain espace V tel que :

$$\forall v \in V,\ \langle B(u), v\rangle - \langle l, v\rangle = 0$$

où B est non linéaire de V dans V', et $l \in V'$, appliquer la méthode de Newton à la résolution de ce problème revient à construire une suite $(u_k)_{k\in\mathbb{N}}$ d'éléments de V définie par $\forall k \in \mathbb{N}$, $u_{k+1} = u_k - h_k$ où h_k est solution du problème variationnel :

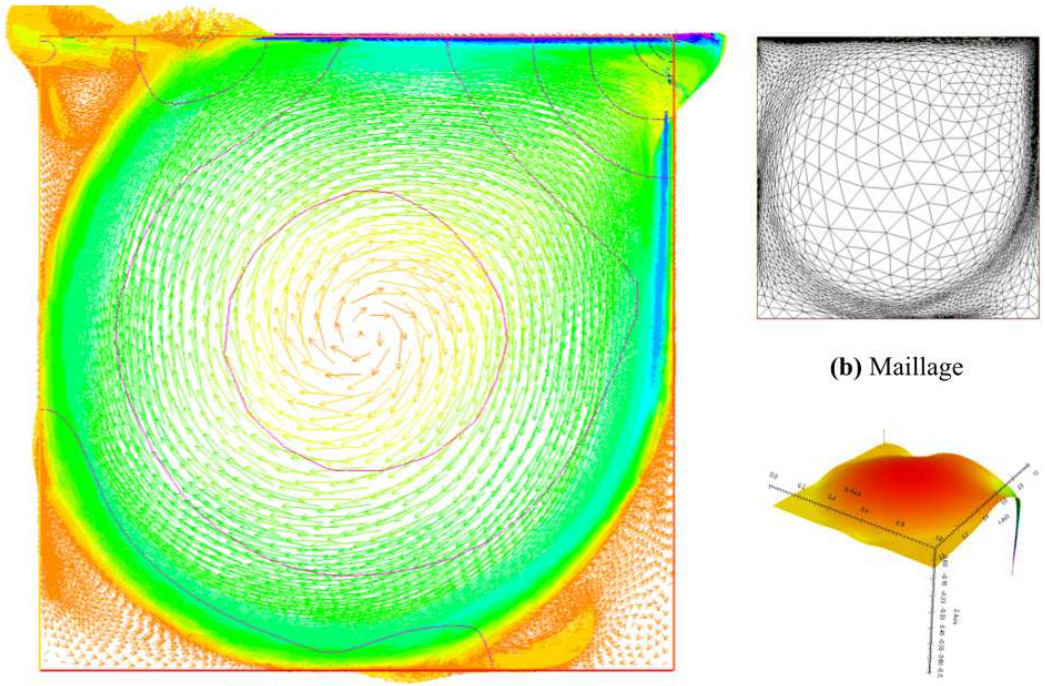
$$\forall v \in V,\ \langle dB(u_k)(h_k), v\rangle = \langle B(u_k), v\rangle - \langle l, v\rangle \tag{1.20}$$

Le problème que l'on résout à chaque itération est lui bien linéaire puisque $dB(u_k)$ désigne une application linéaire de V dans V'. Cette méthode permet, entre autres, de résoudre les équations de Navier-Stokes pour un fluide incompressible et stationnaire, pour des nombres de Reynolds raisonnables, la limite se situant autour de $R_e = 10000$. Pour les calculs avec les Reynolds les plus élevés, plusieurs approximations sont nécessaires, la méthode devant être initialisée par la solution à un Reynolds moindre pour une convergence certaine. Pour les Reynolds les plus bas, $R_e \leq 100$, l'algorithme pourra être initialisé par la solution du problème de Stokes. Nous aborderons au chapitre 3 un problème d'optimisation impliquant la résolution de ces équations selon cette méthode (section 3.3). Quoi qu'il en soit, l'application de la méthode de Newton est soumise à la condition de trouver une initialisation valable, ce qui est en général une tâche difficile.

```
mesh Th = square(15,15);
fespace Wh(Th,[P2,P2,P1]);
int adaptlevel=2,newtonitermax=20;
Wh [ux,uy,p],[vx,vy,q];
//initialisation :
solve Stokes([ux,uy,p],[vx,vy,q]) =           //Système de Stokes
    int2d(Th)(  dx(ux)*dx(vx) + dy(ux)*dy(vx)
              + dx(uy)*dx(vy) + dy(uy)*dy(vy) //∇u : ∇v
              - p*(dx(vx)+dy(vy))             //−p∇.v
              - q*(dx(ux)+dy(uy)) )           //−q∇.u
  //+ on(3,ux=16*(x^2)*(1-x)^2,uy=0) // très régulier
  + on(3,ux=1,uy=0) //très irrégulier
  + on(1,2,4,ux=0,uy=0);

real Re = 10000;//Reynolds ciblé
real R = Re<100 ? Re : 100.;
while(R<=Re)
{
  for(int adapt=0;adapt<adaptlevel;++adapt)
  {
    Th = adaptmesh(Th,[ux,uy],p,err=0.01,hmin=1e-5)
    [ux,uy,p] = [ux,uy,p];

    Wh [hx,hy,r]=[1,1,1];
    int i=0;
    real nu=1./R;
    cout << "Newton method for Re=" << R << endl;
    while(hx[].l2>1.e-8 && i<newtonitermax)
    {
      solve NewtonDirection([hx,hy,r],[vx,vy,q]) =
       int2d(Th)(nu*(dx(hx)*dx(vx)+dy(hx)*dy(vx)+dx(hy)*dx(vy)+dy(hy)*dy(vy))
                 + r*(dx(vx)+dy(vy)) + q*(dx(hx)+dy(hy))
                 + hx*dx(ux)*vx + hy*dy(ux)*vx + hx*dx(uy)*vy + hy*dy(uy)*vy
                 + ux*dx(hx)*vx + uy*dy(hx)*vx + ux*dx(hy)*vy + uy*dy(hy)*vy
                 + r*q*0.0000001)  //Terme de stabilisation
      -int2d(Th)(nu*(dx(ux)*dx(vx)+dy(ux)*dy(vx)+dx(uy)*dx(vy)+dy(uy)*dy(vy))
                 + p*(dx(vx)+dy(vy)) + q*(dx(ux)+dy(uy))
                 + ux*dx(ux)*vx + uy*dy(ux)*vx + ux*dx(uy)*vy + uy*dy(uy)*vy
                 + p*q* 0.0000001)
      +on(1,2,3,4,hx=0,hy=0);//Fin déclaration solde
      cout << "  iter " << i << "  err=" << (hx[].l2) << endl;
      ux[] -= hx[];
      ++i;
    }
  }
  plot([ux,uy],p,cmm="Re="+R);
  R *= 2;
  if(R>Re && R<2*Re) R = Re;
}
```



(a) Champ de vitesse

(b) Maillage

(c) Champ de pression

FIGURE 1.10: Fluide incompressible $R_e = 10000$

```
50 plot([ux,uy],p,cmm="Driven cavity for Re="+Re,wait=1);
```

On trouvera en 3.3 la formulation variationnelle complète des équations de Navier-Stokes, pour le fluide incompressible et dans le cas stationnaire, ainsi que sa différentielle que nous utilisons dans le script ci-dessus, proposé pour la résolution de ce problème. On y remarquera l'utilisation de l'adaptation de maillage, très avantageuse dès que l'on souhaite résoudre ces équations avec un nombre de Reynolds R_e de l'ordre du millier et plus. En raison des nombreuses itérations nécessaires pour aboutir à la solution, ce calcul est plus exigeant que ceux que nous avons rencontrés jusqu'ici, et il faudra compter quelques minutes sur un ordinateur personnel pour le mener à son terme.

Enfin, pour les équations d'évolution dont le caractère non linéaire est dû à la présence d'un terme d'advection, il est possible d'utiliser la méthode des caractéristiques. Citons aussi, pour le cas général, les méthodes de point fixe (desquelles on peut dire que la méthode de Newton est un cas particulier).

1.6.5 Problèmes aux valeurs propres

Dans la version complète, FreeFem++ inclus une interface pour certaines fonctions de la bibliothèque ARPACK (*cf.* [5]). Ces outils sont destinés au calcul des premières valeurs propres pour de grands problèmes à matrice creuse par des méthodes basées sur celle d'Arnoldi [84]. En dehors de certains algorithmes d'optimisation qui utilisent parfois dans leurs mécaniques internes le calcul de valeurs propres, il s'agit d'un domaine de l'algèbre linéaire avec lequel nous n'avons pas été directement confronté au cours de cette thèse.

C'est pourquoi nous nous restreignons ici à la simple mention de cet outil et renvoyons les lecteurs à la documentation de FreeFem++ pour en apprendre plus sur son utilisation dans le logiciel, ou encore aux références indiquées dans cette courte section.

Chapitre 2

L'optimisation dans FreeFem++

Nous présentons dans la première partie de ce chapitre les quelques éléments de théorie qu'il faut connaître pour la pratique de l'optimisation numérique et la compréhension des algorithmes qui seront décrits en seconde partie. Pour notre étude, nous nous sommes inspirés de [16] qui décrit et analyse la plupart des méthodes d'optimisation modernes, et de [77], qui aborde la question de l'optimisation sous contraintes avec une approche plus générale.

Nous utiliserons l'abus de langage, habituellement commis dans la dénomination des problèmes d'optimisation, qui consiste à qualifier un problème de *linéaire* lorsque la fonction coût est une fonction affine. De même que les contraintes seront dites *linéaires* lorsque les fonctions contrainte sont affines. Il est également commun de parler de problème *quadratique* quand la fonction coût est la somme d'une forme quadratique et d'une forme linéaire. Enfin, tout problème n'entrant pas dans l'une des deux catégories précédemment définies sera dit *non-linéaire*.

2.1 Quelques généralités sur l'optimisation

2.1.1 Positionnement du problème

L'énoncé le plus général d'un problème d'optimisation consiste à, étant donnés un ensemble X et une fonction $f : X \to \mathbb{R}$:

$$\textit{trouver } x^* \in X \textit{ tel que } \forall x \in X,\ f(x^*) \leq f(x) \tag{2.1}$$

On appelle f la *fonction coût*, ou encore *fonction objectif* et x^* un *minimiseur* de cette fonction coût. Selon les propriétés de l'ensemble X et de f, cette définition éminemment générale implique une grande variété de méthodes, que ce soit pour établir des résultats d'existence et d'unicité, déterminer le ou les minimiseurs ou encore en donner une caractérisation, émettre des conditions nécessaires ou suffisantes d'optimalité, etc... La nature mathématique de X et les propriétés de la fonction coût f sont également déterminantes pour proposer des méthodes permettant de trouver la ou les solutions de ce problème que l'on ne pourra, le plus souvent, qu'approcher numériquement avec l'aide de l'outil informatique.

Nous nous intéresserons ici plus particulièrement aux algorithmes permettant d'obtenir une telle approximation des optima, locaux ou globaux, dans le cas où X est une partie d'un espace euclidien $\mathbb{R}^n$ et f est une fonction, le plus souvent deux fois dérivable, au moins continue, de X dans $\mathbb{R}$. Il arrivera parfois dans le troisième chapitre que nous énoncions préliminairement certains problèmes sur des espaces de fonctions de dimension

infinie. La plupart des énoncés qui suivront se généralisent à des fonctionnelles définies sur un espace de Banach. Cependant, la motivation première de notre étude étant d'approcher numériquement des solutions de ces problèmes, la résolution effective de ceux-ci portera systématiquement sur les versions discrétisées des objets mathématiques impliqués. Les fonctionnelles minimisées en pratique seront donc toujours définies sur un espace de dimension finie. On parlera d'optimisation *libre* si la recherche de minimiseur se fait sur tout l'espace $\mathbb{R}^n$, et d'optimisation *sous contraintes* lorsque l'on se limite à une partie $X \subset \mathbb{R}^n$. Là encore, il est nécessaire d'imposer une restriction sur les sous domaines dans lesquels la recherche peut être effectuée, qui doivent pouvoir être caractérisés par des propriétés facilement interprétables dans le langage de l'informatique. Le plus souvent, et c'est le cas auquel nous nous limiterons, il s'agira de pouvoir caractériser X à l'aide de m fonctions $c_i : \mathbb{R}^n \to \mathbb{R}$, telles que, étant donné une partition $\{\mathcal{E}, \mathcal{I}\}$ de $[\![1, m]\!]$:

$$X = \{x \in \mathbb{R}^n \mid \forall i \in \mathcal{E}, c_i(x) = 0 \text{ et } \forall i \in \mathcal{I}, c_i(x) \leq 0\}$$

On parlera de *fonctions contraintes* ou tout simplement de *contraintes* pour désigner les fonctions c_i. Les problèmes d'optimisation que nous envisagerons auront donc la forme suivante :

$$\textit{trouver } x^* \in \mathbb{R}^n \textit{ tel que } \begin{cases} \exists V \subset \mathbb{R}^n \textit{ tel que } \forall x \in V,\ f(x^*) \leq f(x) \\ \forall i \in \mathcal{E},\ c_i(x^*) = 0 \\ \forall i \in \mathcal{I},\ c_i(x^*) \leq 0 \end{cases} \tag{2.2}$$

où V est un voisinage de x^*. On écrira alors, de manière condensée : $x^* = \underset{x \in X}{\operatorname{argmin}}\ f(x)$, même si ce minimum n'est pas global, par abus de notation. Pour un problème de ce type, on est capable d'établir des conditions nécessaires ou suffisantes d'optimalité que nous allons aborder dans la suite.

2.1.2 Minimum local non contraint

Dans le cas d'un problème non contraint, l'annulation du gradient en tout optimum local constitue une condition nécessaire puissante dont découlent la plupart des algorithmes d'optimisation libre. Lorsque la fonction coût est deux fois continûment dérivable, on obtient une condition suffisante de minimum local en adjoignant une hypothèse de convexité locale. Certains algorithmes exploitent parfois cette condition suffisante pour se débarrasser des points vérifiant la condition de stationnarité sans pour autant réaliser un minimum ou même un optimum. Ces conditions bien connues se résument dans les propositions suivantes :

Proposition 2.1. *Deux conditions nécessaires :*
- Si f est de classe C^1 et admet un minimum local en x^* alors $\nabla f(x^*) = 0$.
- Si f est de classe C^2 et admet un minimum local en x^* alors $\nabla^2 f(x^*)$ est semi-définie positive.

Proposition 2.2. *Si f est de classe C^2, si $\nabla f(x^*) = 0$ et $\nabla^2 f(x^*)$ est définie positive, alors x^* est un minimum local de f.*

Dans sa version la plus dépouillée, un algorithme d'optimisation sans contrainte consiste en la résolution de l'équation $\nabla f = 0$. Les dérivées secondes peuvent également être exploitées pour éliminer les points en lesquels la hessienne n'est pas définie positive, ou encore pour déterminer des directions de descente robustes comme c'est le cas dans les algorithmes basés sur la méthode de Newton.

2.1.3 Minimisation sous contraintes

Les conditions d'optimalité dans le cas contraint sont plus complexes et nécessitent le plus souvent la restriction du problème par diverses hypothèses techniques. Quoi qu'il en soit, la notion de *multiplicateurs de Lagrange*, introduite par les conditions de Karush-Kuhn-Tucker, joue dans tous les cas un rôle important dans la caractérisation des optima locaux.

Avec les notations de 2.2 et en notant $c = (c_i)_{1\leq i\leq m} : \mathbb{R}^n \to \mathbb{R}^m$, ces conditions sont les suivantes :

Définition 2.3. *On dit qu'un point $x^* \in \mathbb{R}^n$ vérifie les conditions KKT si il existe m réels $(\lambda_1, \ldots, \lambda_m) = \lambda \in \mathbb{R}^m$, appelés multiplicateurs de Lagrange, tels que :*

$$\begin{cases} \nabla f(x^*) - J_c(x^*)^{\mathrm{T}}\,\lambda = 0 & \textit{stationnarité,} \\ \forall i \in \mathcal{I},\ c_i(x^*) \leq 0 \text{ et } \forall j \in \mathcal{E},\ c_j(x^*) = 0 & \textit{faisabilité,} \\ \forall i \in \mathcal{I},\ \lambda_i \geq 0 & \textit{positivité des multiplicateurs et} \\ \sum_{i\in\mathcal{I}} \lambda_i c_i(x^*) = 0 & \textit{complémentarité.} \end{cases}$$

où $J_c(x)$ désigne la matrice jacobienne de l'application c en $x \in \mathbb{R}^n$:

$$J_c(x) = \left[\frac{\partial c_i}{\partial x_j}(x)\right]_{\substack{1\leq i\leq m \\ 1\leq j\leq n}} \in \mathcal{M}_{m,n}(\mathbb{R})$$

dont la i-ème ligne correspond au gradient de c_i, évalué en x.

Dans le cas où le problème ne comporte que des contraintes d'égalité ($\mathcal{I} = \emptyset$ et $\mathcal{E} = [\![1, m]\!]$), la condition nécessaire du premier ordre se simplifie. Il s'agit des conditions dites de Lagrange :

Définition 2.4. *Un point $x^* \in \mathbb{R}^n$ vérifie les conditions de Lagrange si il existe un jeu de multiplicateurs de Lagrange $\lambda \in \mathbb{R}^m$ tel que :*

$$\nabla f(x^*) + \sum_{i=1}^{m} \nabla c_i(x^*)\lambda_i = 0 \quad \textit{et} \quad c(x^*) = 0$$

Pour que la vérification de ces propriétés constitue une condition nécessaire d'optimalité, les contraintes doivent être *qualifiées*, c'est-à-dire que certaines hypothèses techniques doivent être vérifiées en les points candidats à l'optimalité. Par exemple, dans le cas où seules sont présentes des contraintes d'inégalité, on dira de celles-ci qu'elles sont *qualifiées* en un point lorsqu'elles vérifient les conditions de Mangasarian-Fromovitz en ce point (voir [44], [29]) :

Définition 2.5. - Conditions de Mangasarian-Fromovitz : *Les contraintes sont qualifiées en $x \in \mathbb{R}^n$ si :*

$$\forall i, c_i(x) < 0 \textit{ ou } \exists p \in \mathbb{R}^n \mid \nabla c_i(x).p < 0,\ \forall i \textit{ tel que } c_i(x) = 0$$

Dans le cas de problèmes comprenant à la fois des contraintes d'inégalité et d'égalité, en plus des conditions ci-dessus, il est nécessaire que les gradients des contraintes d'égalité en un point donné soient linéairement indépendants pour que 2.3 soit une condition nécessaire d'optimalité en ce point.

Il existe d'autres énoncés de qualification des contraintes. Ces hypothèses sont utiles pour établir en théorie que les conditions 2.3 sont bien des conditions nécessaires d'optimalité. Lorsqu'elles ne sont pas vérifiées, il se peut qu'un optimum soit ou ne soit pas un point vérifiant les conditions KKT. La condition de stationnarité perd alors son utilité pour la recherche d'extrema. On remarquera que si toutes les contraintes d'inégalité sont affines, alors 2.5 est toujours vérifiée. On peut également établir des conditions de qualification non locales dans le cas de fonctions contrainte convexes (voir [29]).

Des conditions suffisantes d'optimalité faisant intervenir les dérivées d'ordre deux des fonctions définissant le problème pourront être trouvées par le lecteur dans [44]. Celles-ci permettent de faire le tri parmi les points vérifiant 2.3 qui ne sont pas des minima locaux.

2.2 La méthode de Newton

2.2.1 Méthode de Newton pour la recherche de racines

Cette méthode occupe une place importante en optimisation moderne car elle est la base de la plupart des méthodes de recherche d'optima efficaces dans le cas de fonctions coût différentiables. A sa source, se trouve la méthode bien connue permettant d'approcher les zéros d'une fonction $f : I \subset \mathbb{R} \to \mathbb{R}$, en construisant la suite $(x_k)_{k\in\mathbb{N}}$ définie par le procédé récursif suivant :

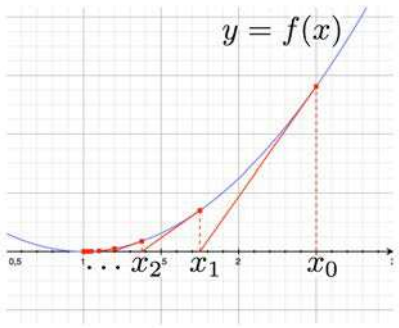


FIGURE 2.1: Représentation graphique de la méthode de Newton pour la recherche d'une racine d'une fonction de $\mathbb{R}$ dans $\mathbb{R}$.

$$\begin{cases} x_0 \in I \\ x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)} \quad , \quad k \in \mathbb{N} \end{cases}$$

Elle se généralise aux applications de $X \subset \mathbb{R}^n$ dans $\mathbb{R}^n$, c'est-à-dire aux systèmes de n équations à n inconnues :

$$\begin{cases} x_0 \in X \\ x_{k+1} = x_k - J_f(x_k)^{-1} f(x_k) \quad , \quad k \in \mathbb{N} \end{cases} \tag{2.3}$$

pour peu que la jacobienne de f soit inversible en chacun des x_k. Le calcul de la mise à jour $J_f(x_k)^{-1} f(x_k)$ constitue l'étape coûteuse de la méthode et l'on peut conserver la même jacobienne pendant plusieurs itérations, et même la remplacer par des approximations : $x_{k+1} = x_k - A_k{}^{-1} f(x_k)$ (voir [29]).

La méthode peut également se généraliser à des fonctions de, et à valeur dans, un espace de Banach. C'est ce que l'on fera dans le chapitre 3 en appliquant la méthode à la résolution des équations de Navier-Stokes dont la formulation variationnelle peut se voir comme la recherche d'un zéro d'une application de V dans son dual V', où V est l'espace de Hilbert des fonctions dans lequel est recherchée la solution.

2.2.2 Application à l'optimisation

Le lien avec l'optimisation est direct puisque, comme nous l'avons vu, la recherche d'un optimum local d'une fonctionnelle se ramène à l'identification d'un point en lequel sa différentielle, ou celle d'un lagrangien, s'annule. Soit $f : \Omega \subset \mathbb{R}^n \to \mathbb{R}$, alors la relation de

récurrence entre les termes de la suite définie par la méthode de Newton pour la résolution de $\nabla f(x) = 0$ s'écrit :

$$x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k)$$

Le succès de cette méthode réside dans son efficacité, avec une convergence quadratique dans les cas les plus favorables [16].

On peut également utiliser la méthode de Newton pour résoudre un problème d'optimisation sous contraintes d'égalité par résolution du système d'équations non linéaires d'inconnues $(x, \lambda) \in \mathbb{R}^n \times \mathbb{R}^m$ constitué par la condition d'optimalité de Lagrange 2.4 en la réinterprétant comme l'annulation de la fonction :

$$\begin{pmatrix} x \\ \lambda \end{pmatrix} \in \mathbb{R}^n \times \mathbb{R}^m \longmapsto \begin{pmatrix} \nabla f(x) + \sum_{i=1}^{m} \lambda_i \nabla c_i(x) \\ c(x) \end{pmatrix} \in \mathbb{R}^n \times \mathbb{R}^m \tag{2.4}$$

Dans ce cas, il faut trouver une initialisation $(x_0, \lambda_0) \in \mathbb{R}^n \times \mathbb{R}^m$ pour le point initial et les multiplicateurs de Lagrange initiaux. A chaque itération, l'itérée suivante est construite en posant $(x_{k+1}, \lambda_{k+1}) = (x_k, \lambda_k) - (h_k, \xi_k)$ où (h_k, ξ_k) est solution du système linéaire suivant :

$$\begin{bmatrix} \nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_{k,i} \nabla^2 c_i(x_k) & J_c(x_k)^{\mathrm{T}} \\ J_c(x_k) & 0 \end{bmatrix} \begin{pmatrix} h_k \\ \xi_k \end{pmatrix} = \begin{pmatrix} \nabla f(x_k) + \sum_{i=1}^{m} \lambda_{k,i} \nabla c_i(x_k) \\ c(x_k) \end{pmatrix} \tag{2.5}$$

Les problèmes avec contraintes d'inégalité sont plus délicats à prendre en main et il n'existe pas d'application aussi directe de la méthode de Newton à ce cas. Les méthodes de points intérieurs décrites dans le chapitre 2.5 constituent un exemple d'adaptation de la méthode de Newton à la résolution de tels problèmes.

2.2.3 Difficultés liées à l'initialisation

Obtenir une situation de convergence favorable n'est pas nécessairement simple et c'est là l'une des principales difficultés posée par la méthode. La fractale dite de Newton constitue une illustration intéressante de l'imprévisibilité de la convergence par rapport à l'initialisation de la suite. Celle-ci se construit en analysant le comportement de la méthode de Newton appliquée à une fonction méromorphe, le plus souvent un polynôme de la forme $f(z) = z^p - 1$. Selon le nombre complexe z_0 choisi pour l'initialisation, la méthode de Newton $z_{n+1} = z_n - f(z_n)/f'(z_n)$ va soit converger vers l'une des racines p-ième de l'unité, soit ne pas converger. En associant une couleur selon ces comportements asymptotiques, on obtient des images d'une grande complexité qui montrent bien le comportement erratique de la méthode : des points infiniment proches peuvent converger vers des racines différentes (figure 2.2).

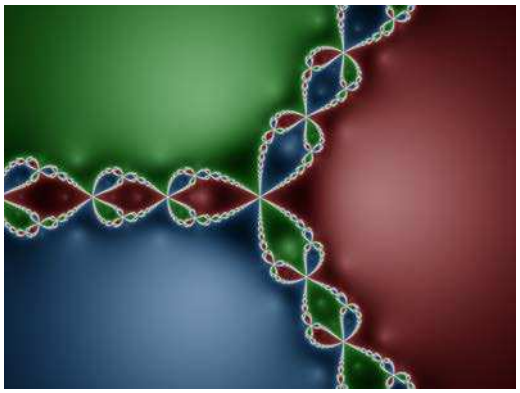


Figure 2.2: Fractale de Newton du troisième ordre

Dans la pratique, ce sont essentiellement les cas où l'approximation ne converge pas qui sont

le plus redoutés et les raffinements modernes dont la méthode est enrichie consistent le plus souvent en l'adjonction d'une phase permettant d'éviter cette divergence. On parle alors de convergence *globale*, dans le sens où, quelle que soit l'initialisation, la méthode converge vers une limite, par opposition aux hypothèses sous lesquelles les théorèmes de convergence ([29], [16]) pour la méthode de Newton s'appliquent, qui demandent que le terme initial appartienne au voisinage d'un minimum. Dans le cadre de l'optimisation, cette convergence globale peut être obtenue en ajoutant la possibilité à chaque itération de déterminer un pas minimisant la fonctionnelle le long de la direction obtenue par le calcul de $\left[\nabla^2 f(x)\right]^{-1} f(x)$. Cette phase de recherche linéaire permet de ramener l'itérée dans l'un des bassins d'attraction de la fonctionnelle et donc d'éviter une issue fatale.

2.3 Recherches linéaires

Un cas particulier important, bien que nous ne l'aborderons pas en détail dans cette thèse, est celui de la minimisation des fonctions d'une unique variable. On a en effet vu que la plupart des algorithmes utilisant les dérivées de la fonctionnelle comportent à chaque itération une phase consistant à, après avoir déterminé une direction de recherche $d_k \in \mathbb{R}^n$, trouver la valeur $t_k \in \mathbb{R}$ réalisant le minimum de la fonction $t \mapsto f(x_k + t d_k)$ afin de déterminer l'itérée suivante $x_{k+1} = x_k + t_k d_k$. La stabilité d'un algorithme à direction de descente est très dépendante de la recherche linéaire utilisée, c'est pourquoi il est crucial de disposer de méthodes très robustes pour cette tâche.

L'approche classique pour l'adaptation d'une recherche linéaire aux problèmes avec contraintes est de l'appliquer a une fonction de mérite pénalisant une mesure de la violation de la contrainte. On pourra à ce sujet se référer au paragraphe 2.4 pour des suggestions de fonctions de mérite. Cette méthode suppose qu'il est possible d'évaluer la fonction coût en des points qui ne vérifient pas les contraintes.

Une autre technique pour la recherche des pas optimaux pour des problèmes sous contraintes est la méthode de *filtre*. Elle se base sur des arguments développés dans le cadre de l'optimisation multi-objectifs. Cette méthode est en particulier employée dans le logiciel IPOPT que nous présentons dans la section 2.5.3, dans laquelle nous aborderons les principales idées de cette notion de filtre. Là encore, la fonction coût doit être définie en des points ne vérifiant pas strictement les contraintes.

2.4 NLopt et le lagrangien augmenté

NLopt est une bibliothèque open-source compilant plusieurs logiciels d'optimisation sous une interface commune. Elle propose quelques algorithmes d'optimisation stochastique ainsi qu'un choix assez vaste de méthodes classiques utilisant ou non les dérivées. L'intérêt de cette bibliothèque est surtout de permettre l'optimisation sous contraintes par le biais de méthodes dédiées, mais également par la possibilité d'encapsuler n'importe lequel des optimiseurs dans un formalisme de lagrangien augmenté, ce qui multiplie les solutions possibles pour traiter les problèmes d'optimisation sous contraintes.

Interfacés dans FreeFem++, ces algorithmes sont très faciles à utiliser et offrent des solutions rapides pour des problèmes d'optimisation en petite à moyenne dimension. Les implémentations des algorithmes les plus efficaces de la bibliothèques (BFGS, Truncated Newton, NEWUOA, etc...) utilisent là encore des matrices pleines de dimension de l'ordre de grandeur du nombre de variables d'optimisation, ce qui compromet leur utilisation pour les problèmes avec un grand nombre de paramètres.

Nous ne détaillerons pas les mécanismes mathématiques de chacun des algorithmes de NLopt. Le nombre de ces méthodes est bien trop grand pour qu'une telle description trouve sa place dans ce manuscrit. Il est par contre peut-être utile de préciser quelque peu le fonctionnement du lagrangien augmenté, puisqu'il s'agit de la méthode potentiellement la plus intéressante qu'introduit la bibliothèque, et qu'elle peut s'appliquer à n'importe laquelle des routines de l'interface.

2.4.1 Lagrangien augmenté : un bref aperçu

Les méthodes de *lagrangien augmenté* consistent à construire une fonction de mérite en ajoutant un terme de pénalisation au lagrangien associé au problème d'optimisation sous contraintes. On reprend les notations de la définition 2.2, auxquelles on ajoute des bornes simples, de sorte à ce que le domaine d'optimisation se limite aux points $x \in \mathbb{R}^n$ tels que $x_l \leq x \leq x_u$ (où $x_l,\ x_u \in \mathbb{R}^n$ et les inégalités sont vérifiées composante par composante). Considérons la fonction de mérite suivante, sans se soucier des bornes x_l et x_u dans un premier temps :

$$\Phi(x, \lambda; \mu) = \underbrace{f(x) - \sum_{i=1}^{m} \lambda_i c_i(x)}_{\mathcal{L}(x,\lambda)} + \frac{\mu}{2} \sum_{i \in \mathcal{E}} c_i(x)^2 + \frac{\mu}{2} \sum_{i \in \mathcal{I}} c_i^+(x)^2 \tag{2.6}$$

La présence de contraintes d'inégalité conduirait cependant à une perte de dérivabilité puisque la fonction $c_i^+ : x \mapsto \max(0, c_i(x))$ n'est en général pas de classe C^2. Ce serait alors gênant car ce type de contraintes rendrait impossible l'utilisation de méthodes d'ordre élevé ou approchant des objets d'ordre élevé. Les contraintes d'inégalité sont donc artificiellement transformées en contraintes d'égalité par l'introduction de variables additionnelles $s_i, i \in \mathcal{I}$, appelées en anglais *slack variables*, lesquelles seront simplement contraintes par une borne fixe. Les contraintes d'inégalité se reformulent ainsi :

$$c_i(x) + s_i = 0, \quad s_i \leq 0, \quad \forall i \in \mathcal{I} \tag{2.7}$$

Cette astuce permet l'utilisation d'une fonction de mérite conservant les propriétés de différentiation de f :

$$\Phi(x, \lambda, \mu) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x) + \frac{\mu}{2} \sum_{i=1}^{m} c_i(x)^2 \tag{2.8}$$

L'avantage d'introduire les multiplicateurs de Lagrange dans la fonction de mérite, en plus du terme de pénalisation, est que ceux-ci permettent la réalisation des contraintes sans atteindre des valeurs de μ disproportionnées (voir [77]). On évite ainsi l'apparition de systèmes linéaires mal conditionnés.

La méthode de lagrangien augmenté consiste donc à établir une séquence de valeurs de λ et μ en déterminant pour chacune d'elles le minimiseur :

$$\underset{x \in \mathbb{R}^n \ \mid \ x_l \leq x \leq x_u}{\operatorname{argmin}} \quad \Phi(x, \lambda, \mu) \tag{2.9}$$

D'après [77], la condition KKT que doit vérifier le minimiseur x pour un tel problème s'écrit :

$$x - P\left(x - \nabla\Phi(x, \lambda, \mu), x_l, x_u\right) = 0 \tag{2.10}$$

où $P(v,l,u),\ v,l,u \in \mathbb{R}^n$ est la projection du vecteur u dans l'hyper-rectangle $\{x \in \mathbb{R}^n \mid l_i \leq x_i \leq u_i,\ \forall i \in [\![1,n]\!]\}$, telle que la composante $i, 1 \leq i \leq n$ de $P(v,l,u)$ vérifie :

$$P(v,l,u)_i = \begin{cases} l_i & \text{if } v_i \leq l_i \\ v_i & \text{if } v_i \in [l_i, u_i] \\ u_i & \text{if } v_i \geq u_i \end{cases}$$

Le paramètre μ suit une progression croissante et doit *a priori* devenir arbitrairement grand. Lorsque c'est possible, les composantes de λ doivent quant à elles converger vers un jeu de multiplicateurs de Lagrange convenable pour les conditions KKT, aussi sont-elles modifiées de manière à satisfaire cette exigence. On proposera, très classiquement, l'algorithme suivant pour la résolution de ce problème de minimisation par la méthode du lagrangien augmenté :

Algorithm 1 Un algorithme simple pour le lagrangien augmenté

Initialisation : *Choisir un point initial $x_0 \in \mathbb{R}^n$ ainsi que des multiplicateurs de Lagrange initiaux $\lambda_0 \in \mathbb{R}^m$, les valeurs des tolérances pour établir les critères de convergence η^* et ω^*, et un éventuel nombre d'itération maximal N.*
Initialiser les scalaires $\mu_0 = 10,\ \omega_0 = 1/\mu_0$ et $\eta_0 = 1/\mu_0^{0.1}$.

---

for k=0 **to** k=N **do**
  Trouver x_k tel que $\|x_k - P\left(x_k - \nabla\Phi(x_k, , \lambda_k, \mu_k), x_l, x_u\right)\| \leq \omega_k$
  if $\|c(x_k)\| \leq \eta_k$ **then**
    if $\|c(x_k)\| \leq \eta^*$ **and** $\|x_k - P\left(x_k - \nabla\Phi(x_k, , \lambda_k, \mu_k), x_l, x_u\right)\| \leq \omega^*$ **then**
      Fin de l'algorithme avec la solution x_k
    end if
    Mise à jour des multiplicateurs de Lagrange :
      $\lambda_{k+1} = \lambda_k - \mu_k c(x_k)$
    Réduction des tolérances :
      $\mu_{k+1} = \mu_k$
      $\eta_{k+1} = \eta_k / \mu_{k+1}^{0.9}$
      $\omega_{k+1} = \omega_k / \mu_{k+1}$
  else
    Augmentation du paramètre de pénalisation et diminution des tolérances :
      $\lambda_{k+1} = \lambda_k$
      $\mu_{k+1} = 100\mu_k$
      $\eta_{k+1} = \eta_k / \mu_{k+1}^{0.1}$
      $\omega_{k+1} = 1/\mu_{k+1}$
  end if
end for

Dans l'algorithme 1, le vecteur x_k tel que $\|x_k - P\left(x_k - \nabla\Phi(x_k, , \lambda_k, \mu_k), x_l, x_u\right)\| \leq \omega_k$ est obtenu par résolution de 2.9, c'est-à-dire par minimisation de $x \mapsto \Phi(x, \lambda_k, \mu_k)$. Il est donc nécessaire d'utiliser un algorithme d'optimisation libre secondaire pour accomplir cette tâche. Dans la bibliothèque NLopt, l'utilisateur peut choisir n'importe lequel des autres algorithmes proposés, ce qui permet de résoudre des problèmes d'optimisation sous contraintes avec des algorithmes *a priori* inutilisables dans ce cas, à la condition que la fonctionnelle soit définie pour des valeurs de x violant les contraintes. Une version du lagrangien augmenté spécialisée dans le traitement des contraintes d'égalité est également proposée. Lors de son utilisation, la possibilité de traiter des problèmes avec contraintes

d'inégalité est alors soumise à l'adéquation de l'algorithme secondaire avec ce type de problème.

Les implémentations effectivement proposées dans la bibliothèque NLopt diffèrent de l'algorithme 1, dont le but n'est que de donner une illustration très simple et générale des méthodes de lagrangien augmenté dans le cas où les deux types de contraintes sont présents. On pourra trouver des descriptions plus exactes dans la littérature proposée dans [60], et notamment dans [15] dont les implémentations de NLopt semblent être largement tributaires, ainsi que dans [33].

2.4.2 L'interface FreeFem++ pour NLopt

Les interfaces pour les routines de NLopt ont été regroupées dans le même paquet `ff-NLopt`. La liste exhaustive des algorithmes figure dans la table 2.3, dans laquelle apparaissent également quelques éléments essentiels pour l'écriture d'un script FreeFem++ appelant l'une des routines associées. Le lecteur est renvoyé à [60] pour de plus amples précisions, ainsi que pour toutes considérations d'ordre technique, aussi bien sur le plan informatique que mathématique. Comme nous l'avons déjà indiqué, ces algorithmes sont plus adaptés à la résolution de problèmes avec un petit nombre de variables, pour lesquels l'utilisation de matrices pleines se révèle plus ergonomique.

L'appel à une routine de NLopt dans un script FreeFem++ se fait par la commande suivante :

```
load "ff-NLopt"
... //définition de J, u, grad, etc...
real min = nloptXXXXXX(J,u,   //partie obligatoire
                       grad = <identifiant de grad> , //si nécessaire
                       lb = //bornes inférieures x_l
                       ub= //bornes supérieures x_u
                       ...//les arguments optionnels :
                          //contraintes non-linéaires
                          //critères d'arrêt,
                          //paramètres spécifiques à l'algorithme
                          //etc...
                       );
```

Dans cet exemple d'appel, `XXXXXX` désigne le nom de l'algorithme qui apparaît dans la première colonne du tableau 2.3, `u` est le point en lequel est initialisé l'algorithme, de type `real[int]` et qui contiendra le résultat de l'optimisation à la fin de la méthode. Comme avec les précédentes routines d'optimisation, la fonction coût `J` doit avoir le prototype `real (real[int] &)`. Selon l'algorithme utilisé, les paramètres `grad`, `lb` et `ub` peuvent être obligatoires, optionnels ou inutiles. Le tableau 2.3 permet de savoir quel algorithme nécessite une implémentation du gradient de la fonction coût, s'il est nécessaire de spécifier des bornes ou encore quelle méthode supporte quel type de contraintes, etc... Ce document ne saurait se substituer au tutoriel d'utilisation de la documentation de FreeFem++ [79] à laquelle nous renvoyons le lecteur désirant en apprendre plus sur ce sujet, et notamment pour ce qui est des paramètres optionnels ou pour tout autre détail technique en rapport avec l'utilisation de cette interface.

Id Tag	Full Name	Bounds	Gradient -Based	Stochastic	Constraints		Sub-Opt
					Equality	Inequality	
DIRECT	Dividing rectangles	●					
DIRECTL	Locally biased dividing rectangles	●					
DIRECTLRand	Randomized locally biased dividing rectangles	●					
DIRECTNoScal	Dividing rectangles - no scaling	●					
DIRECTLNoScal	Locally biased dividing rectangles - no scaling	●					
DIRECTLRandNoScal	Randomized locally biased dividing rectangles - no scaling	●					
OrigDIRECT	Original Glabonsky's dividing rectangles	●				✔	
OrigDIRECTL	Original Glabonsky's locally biased dividing rectangles	●				✔	
StoGO	Stochastic(?) Global Optimization	●	●				
StoGORand	Randomized Stochastic(?) Global Optimization	●	●				
LBFGS	Low-storage BFGS		●				
PRAXIS	Principal AXIS	✔					
Var1	Rank-1 shifted limited-memory variable-metric		●				
Var2	Rank-2 shifted limited-memory variable-metric		●				
TNewton	Truncated Newton		●				
TNewtonRestart	Steepest descent restarting truncated Newton		●				
TNewtonPrecond	BFGS preconditionned truncated Newton		●				
TNewtonRestartPrecond	BFGS preconditionned truncated Newton with steepest descent restarting		●				
CRS2	Controled random search with local mutation	✔		●			
MMA	Method of moving asymptots	✔	●			✔	
COBYLA	Constrained optimization by linear approximations	✔			✔	✔	
NEWUOA	NEWUOA						
NEWUOABound	NEWUOA for bounded optimization	✔					
NelderMead	Nelder-Mead simplex	✔					
Sbplx	Subplex	✔					
BOBYQA	BOBYQA	✔					
ISRES	Improved stochastic ranking evolution strategy	✔		●	✔	✔	
SLSQP	Sequential least-square quadratic programming	✔	●		✔	✔	
MLSL	Multi-level single-linkage	✔	●/✔	●			●
MLSLLDS	Low discrepancy multi-level single-linkage	✔	●/✔	●			●
AUGLAG	Constraints augmented lagrangian	✔	●/✔		✔	✔	●
AUGLAGEQ	Equality constraints augmented lagrangian	✔	●/✔		✔	✔	●

Legend :

- ✔ Supported and optional
- ✔ Should be supported and optional, may lead to weird behaviour though.
- ● Intrinsic characteristic of the algorithm which then need one or more unavoidable parameter to work (for stochastic algorithm, the population size always have a default value, they will then work if it is ommited)
- ●/✔ For routines with subsidiary algorithms only, indicates that the corresponding feature will depend on the chosen sub-optimizer.

FIGURE 2.3: Liste des algorithmes de NLopt bénéficiant d'une interface dans FreeFem++

2.5 IPOPT et les méthodes de points intérieurs

Les méthodes de *points intérieurs* constituent un raffinement des méthodes de *barrière* qui consiste à minimiser par la méthode de Newton une suite de fonctions de mérite. Ces fonctions de mérite sont construites en ajoutant à la fonctionnelle initiale des termes qui pénalisent les contraintes d'inégalité et tendent vers l'infini au voisinage des points en lesquels celles-ci s'annulent. Les contraintes d'égalité y sont quant à elles traitées par la classique introduction de multiplicateurs de Lagrange. Nous commencerons par une description rapide de ces méthodes de barrière avant de préciser comment les méthodes de points intérieurs en contournent les principales difficultés. Cette étude se base essentiellement sur [44] et, dans une moindre mesure sur [16]. Nous aborderons enfin le cas plus spécifique du logiciel d'optimisation open source IPOPT par une brève énumération des idées les plus importantes de [92].

Pour ce court exposé des méthodes de points intérieurs, nous reformulons 2.2 comme dans [92] en ramenant la définition générale d'un problème d'optimisation sous contrainte à la détermination d'un élément $x^* \in \mathbb{R}^n$ vérifiant :

$$x^* = \underset{x \in V}{\operatorname{argmin}}\ f(x) \text{ avec } V = \{x \in \mathbb{R}^n\ ;\ c(x) = 0 \text{ et } x_l \leq x \leq x_u\} \tag{2.11}$$

où $x_l \in [-\infty, +\infty[^n$ et $x_u \in]-\infty, +\infty]^n$, avec $x_l \leq x_u$, l'inégalité se faisant composante par composante : $\forall x, y \in \mathbb{R}^n,\ x \leq y \ \Leftrightarrow\ \forall i \in [\![1, n]\!], x_i \leq y_i$. Les contraintes d'inégalité sont ici aussi transformées en contraintes d'égalité par l'introduction de *slack variables* (voir 2.7) et l'on suppose que c est une fonction de $\mathbb{R}^n \to \mathbb{R}^m$. Comme il est d'usage dans la littérature spécialisée, pour un vecteur $a \in \mathbb{R}^n$, nous noterons $A = (a_i \delta_{ij})_{1 \leq i,j \leq n}$ la matrice diagonale d'éléments diagonaux a_i et e désignera le vecteur de $\mathbb{R}^n$ dont toutes les composantes sont égales à 1 : $e = (1, 1, \ldots, 1) \in \mathbb{R}^n$.

Avec la formulation 2.11 ci-dessus, en distinguant les multiplicateurs de Lagrange λ associés aux contraintes d'égalité, de ceux correspondant aux simples bornes et que nous noterons respectivement z_l et z_u ($\in \mathbb{R}^n$), les conditions KKT 2.3 s'écrivent plus spécifiquement :

$$\begin{cases} \nabla f(x) + J_c(x)^{\mathrm{T}} \lambda + z_u - z_l = 0 \\ (x_u - x).z_u + (x - x_l).z_l = 0 \\ c(x) = 0 \text{ et } x_l \leq x \leq x_u \\ z_u \geq 0 \text{ et } z_l \geq 0 \end{cases} \tag{2.12}$$

On suppose ici, pour ne pas alourdir les notations, que chacune des composantes des vecteurs bornes x_l et x_u est finie. Dans le cas contraire les termes correspondant aux bornes infinies doivent être retirés dans la condition de complémentarité ci-dessus (seconde équation).

2.5.1 Méthodes de barrière

Les méthodes de barrière permettent la prise en compte des contraintes d'inégalité pour des problèmes de minimisation dans lesquels la vérification de ces contraintes est cruciale puisque, par construction, les variables d'optimisation vérifieront systématiquement ces contraintes tout au long du processus d'optimisation. Une fonction de mérite est conçue en ajoutant à f une fonction *aussi proche que possible* de la fonction nulle dans le domaine de $\mathbb{R}^n$ dont les points vérifient les contraintes, et qui tend vers l'infini lorsqu'on s'approche de la frontière de ce domaine. Cela se réalise en pratique par le choix d'une fonction $I \in C^2(]0, +\infty[, \mathbb{R})$ telle que $I(t) \xrightarrow[t \to 0^+]{} +\infty$, qui permet de construire la fonction de mérite $B : \mathbb{R}^n \times \mathbb{R}_+ \longrightarrow \mathbb{R}$:

$$B(x,\mu) = f(x) + \mu\sum_{i=1}^{n} I(x_{u,i} - x_i) + \mu\sum_{i=1}^{n} I(x_i - x_{l,i}) \tag{2.13}$$

Il serait certainement plus naturel de ne pas occulter les contraintes d'inégalité en les réduisant à de simples bornes avec les *slack variables*, leur introduction directe dans la fonction de mérite par composition avec la fonction I constituant l'essence même des méthodes de points intérieurs :

$$B(x,\mu) = f(x) + \sum_{i\in\mathcal{I}} (I \circ c_i)(x)$$

C'est cependant l'orientation qui a été choisie par les concepteurs de IPOPT et, afin de restituer une description aussi exacte que possible des mathématiques qui sous-tendent les mécaniques internes du logiciel, il nous est nécessaire de conserver cette formulation. On pourra retrouver une introduction plus traditionnelle à vocation didactique dans le très complet [44].

La fonction I porte le nom de *barrier function* dans la littérature anglo-saxonne, que nous traduirons par le terme de *fonction obstacle*. Lorsque cette méthode a été introduite, dans les années 1960, la première fonction obstacle à avoir été proposée fut la fonction inverse, rapidement abandonnée au profit de la fonction logarithme :

$$B(x,\mu) = f(x) - \mu\sum_{i=1}^{n} \ln(x_{u,i} - x_i) - \mu\sum_{i=1}^{n} \ln(x_i - x_{l,i}) \tag{2.14}$$

On choisit alors une suite positive $(\mu_k)_{k\in\mathbb{N}^*}$ décroissant rapidement vers 0 et, pour chaque k, on résout le problème de minimisation sous contrainte :

$$x_k = \underset{x\in\mathbb{R}^n \mid c(x)=0}{\operatorname{argmin}} B(x,\mu_k) \tag{2.15}$$

Ces problèmes d'optimisation ne comportent que des contraintes d'égalité et la condition nécessaire d'optimalité se traduit par l'existence de multiplicateurs de Lagrange $\lambda_k \in \mathbb{R}^m$ vérifiant :

$$\begin{cases} \nabla B(x_k,\mu_k) + J_c(x_k)^{\mathrm{T}}\lambda_k &= 0 \\ c(x_k) &= 0 \end{cases} \tag{2.16}$$

La recherche du couple (x_k, λ_k) est effectuée par la résolution de ce système d'équations non-linéaire par la méthode de Newton. Ainsi, pour une valeur du paramètre μ fixée, le calcul des directions $(d_x, d_\lambda) \in \mathbb{R}^n \times \mathbb{R}^m$ à chaque itération de l'algorithme de Newton se fera par la résolution de systèmes linéaires tels que :

$$\begin{bmatrix} \nabla^2 B(x,\mu) + \sum_{i=1}^{m} \nabla^2 c_i(x)\lambda_i & J_c(x)^{\mathrm{T}} \\ J_c(x) & 0 \end{bmatrix} \begin{pmatrix} d_x \\ d_\lambda \end{pmatrix} = \begin{pmatrix} \nabla B(x,\mu) + J_c(x)^{\mathrm{T}}\lambda \\ c(x) \end{pmatrix} \tag{2.17}$$

Pour les valeurs de μ les plus petites, et dans les cas où le minimiseur active l'une des bornes, on peut s'attendre à obtenir des systèmes mal conditionnés. En effet, selon [44], sous certaines hypothèses nécessaires pour établir la convergence de l'algorithme, la convergence vers le minimiseur x^* est $\|x_\mu - x^*\| = O(\mu)$, et les coefficients de la hessienne de B s'écrivent :

$$\frac{\partial^2 B}{\partial x_i \partial x_j}(x,\mu) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x) + \mu\delta_{ij}\left(\frac{1}{(x_i - x_{l,i})^2} + \frac{1}{(x_{u,i} - x_i)^2}\right) \tag{2.18}$$

Il se peut donc que ces coefficients divergent grossièrement dans les cas d'activation des contraintes.

Cette méthode n'est donc efficace que lorsque le minimiseur vérifie fortement les contraintes, ce qui est plutôt gênant. Elle n'a donc pas rencontré un grand succès avant la mise au point des méthodes de points intérieurs qui contournent cette limitation et offrent un ensemble de méthodes efficaces.

2.5.2 Méthode de points intérieurs primale-duale

Afin d'éliminer les différences colossales entre les ordres de grandeur de certains des termes figurant dans l'expression des coefficients de la matrice du système 2.17, on introduit de nouvelles variables s'apparentant aux multiplicateurs de Lagrange associés aux contraintes d'inégalités. Pour une valeur de μ donnée, au lieu de résoudre directement le système 2.16, on introduit les variables z_l et z_u, toutes deux appartenant à $\mathbb{R}^n$, et l'on résout le système d'équations non-linéaire :

$$S(\mu) : \left\{ \begin{array}{rcl} \nabla f(x) + J_c(x)^{\mathrm{T}} \lambda + z_u - z_l & = & 0 \\ c(x) & = & 0 \\ (X_u - X) z_u - \mu e & = & 0 \\ (X - X_l) z_l - \mu e & = & 0 \end{array} \right. \tag{2.19}$$

On remarquera les similarités que ces égalités entretiennent avec 2.12 lorsque l'on fait tendre μ vers 0.

Les systèmes linéaires pour la recherche des directions de Newton ne comportent désormais plus de quantités potentiellement non bornées quand μ devient petit :

$$\begin{bmatrix} \nabla^2 \mathcal{L}(x,\lambda) & J_c(x)^{\mathrm{T}} & I & -I \\ J_c(x) & 0 & 0 & 0 \\ -Z_u & 0 & X_u - X & 0 \\ Z_l & 0 & 0 & X - X_l \end{bmatrix} \begin{pmatrix} d_x \\ d_\lambda \\ d_u \\ d_l \end{pmatrix} = - \begin{pmatrix} \nabla \mathcal{L}(x,\lambda) + z_u - z_l \\ c(x) \\ (X_u - X) z_u - \mu e \\ (X - X_l) z_l - \mu e \end{pmatrix} \tag{2.20}$$

où $\mathcal{L}(x,\lambda) = f(x) + \sum_{i=1}^{m} c_i \lambda_i$, et où les dérivations pour ce lagrangien qui apparaissent dans ce système ne portent que sur la variable x. Le signe $-$ dans le second membre est une commodité permettant d'écrire l'actualisation des itérés avec une addition plutôt qu'une soustraction, ce qui permet d'avoir des pas positifs lorsque l'on utilise une recherche linéaire.

Pour la résolution de 2.19, une méthode de Newton standard se limiterait à l'algorithme 2 sans prendre en compte de pas : on retrancherait la solution de 2.20 à l'itérée courante, puis on recommencerait ce procédé au nouveau point obtenu jusqu'à arriver à une valeur de l'erreur 2.21 satisfaisante. En théorie, la différence entre μ_k et μ_{k+1} est suffisamment petite pour que le résultat de la résolution de $S(\mu_k)$ constitue un bon terme initial pour celle de $S(\mu_{j+1})$. Mais en pratique, on cherche à réduire autant que possible le nombre de sous-problèmes. Aussi est-il nécessaire d'enrichir la méthode de Newton par des procédés permettant une convergence globale. Cela peut par exemple être réalisé en ajoutant une phase de recherche linéaire pour la détermination d'un pas en plus de la direction dans la méthode de Newton. Nous renvoyons le lecteur à [44] et [16] pour plus d'informations à propos de ces améliorations, nous limitant à énoncer les stratégies implémentées dans IPOPT. Les matrices impliquées dans le système 2.20 sont très creuses. On pourra donc utiliser des solveurs linéaires adaptés à ce type de problème. Il est enfin important de remarquer que l'algorithme de Newton peut également être remplacé sans difficulté par

une méthode de type quasi-Newton telle que BFGS. Il est donc en pratique possible d'utiliser une méthode de points intérieurs sans nécessairement disposer d'une implémentation calculant les dérivées au second ordre de la fonctionnelle et des contraintes, bien que l'on perdre alors le caractère creux du système, ainsi qu'en vitesse de convergence.

Algorithm 2 Méthode de points intérieurs

Initialisations : $x_0 \in \mathbb{R}^n$, $\lambda_0 \in \mathbb{R}^m$, z_0^l *et* z_0^u
Choisir une suite décroissante (μ_k) *convergeant suffisamment vite vers 0 afin que la précision machine soit atteinte après une dizaine de termes.*
Choisir des critères d'arrêt C_k *pour les méthodes de Newton.*
Initialiser toutes les quantités nécessaires dans le cas de l'utilisation d'une méthode quasi-Newton
$k \leftarrow 0$

while $\mu_k > \epsilon$ **do**
  Initialisation de l'algorithme de Newton :
  $j \leftarrow 0$ *(compteur des itérations de l'algorithme de Newton)*
  $x_{k,0} \leftarrow x_k \qquad \lambda_{k,0} \leftarrow \lambda_k$
  $z^l_{k,0} \leftarrow z^l_k \qquad z^u_{k,0} \leftarrow z^u_k$
  while C_k n'est pas rempli **do**
    Trouver $(d^x_j, d^\lambda_j, d^l_j, d^u_j) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^n \times \mathbb{R}^n$ *vérifiant le système 2.20 évalué en* $(x_{k,j}, \lambda_{k,j}, z^l_{k,j}, z^u_{k,j})$.
    Déterminer les pas $\alpha^x_j,\ \alpha^\lambda_j,\ \alpha^l_j,\ \alpha^u_j \in \mathbb{R}$ *par recherches linéaires.*
    $x_{k,j+1} \leftarrow x_{k,j} + \alpha^x_j d^x_j \qquad \lambda_{k,j+1} \leftarrow \lambda_{k,j} + \alpha^\lambda_j d^\lambda_j$
    $z^l_{k,j+1} \leftarrow z^l_{k,j} + \alpha^l_j d^l_j \qquad z^u_{k,j+1} \leftarrow z^u_{k,j} + \alpha^u_j d^u_j$
    $j \leftarrow j + 1$
  end while
  $x_{k+1} \leftarrow x_{k,j} \qquad \lambda_{k+1} \leftarrow \lambda_{k,j}$
  $z^l_{k+1} \leftarrow z^l_{k,j} \qquad z^u_{k+1} \leftarrow z^u_{k,j}$
  $k \leftarrow k + 1$
end while

On trouvera des éléments en rapport avec la convergence des méthodes de points intérieurs dans [16] pour les cas linéaires et quadratiques, et dans [44] pour le cas général. Dans les versions les plus strictes, les méthodes de points intérieurs présentent des caractéristiques similaires à celles de type Newton et apparentés, avec convergence quadratique lorsque l'initialisation de l'algorithme est suffisamment proche d'une solution. En résumé, si l'on note (x^*_k, λ^*_k) une solution de $S(\mu_k)$ (2.19) pour tout k, alors, sous certaines hypothèses, la limite (x^*, λ^*) de (x^*_k, λ^*_k) lorsque k tend vers l'infini vérifie les conditions de Karush-Kuhn-Tucker 2.12.

2.5.3 IPOPT

IPOPT est un logiciel d'optimisation sous contraintes à la pointe de ce qui se fait actuellement dans le domaine de l'optimisation dérivable. Il implémente une méthode de points intérieurs couplée à une recherche linéaire avec *filtre*. Les matrices sont stockées sous forme creuse afin de permettre la résolution de problèmes de grande dimension.

On se place ici dans le cadre de la résolution de 2.12, avec les mêmes notations. Nous nous limitons à une compréhension globale des spécificités d'IPOPT, renvoyant les lecteurs

qui auraient besoin de plus amples explications à l'article [92], ou encore au site web du logiciel [58]. On trouvera également dans l'annexe A.3 le listing étape par étape de l'algorithme.

Critères d'arrêt globaux

En tant que méthode de points intérieurs, IPOPT essaie de construire une approximation pour une solution de 2.12 en appliquant une méthode similaire à l'algorithme 2. Le critère d'arrêt le plus global, en dehors des critères arbitraires tels que les limites sur le nombre d'itérations ou en termes de temps de calcul, porte sur la réduction de la norme infinie des vecteurs apparaissant dans les conditions de Karush-Kuhn-Tucker. Pour une valeur de μ donnée, cette erreur s'exprime comme :

$$E_\mu(x,\lambda,z_u,z_l) = \max\left\{\frac{\left\|\nabla f(x) + J_c(x)^{\mathrm{T}}\lambda + z_u - z_l\right\|_\infty}{s_d}, \|c(x)\|_\infty, \frac{\|(X_u - X)z_u - \mu e\|_\infty}{s_c}, \frac{\|(X - X_l)z_l - \mu e\|_\infty}{s_c}\right\} \tag{2.21}$$

où s_c et s_d sont des coefficients de scaling. L'algorithme prend fin dès qu'un itéré $(\tilde{x}^*, \tilde{\lambda}^*, \tilde{z}_u^*, \tilde{z}_l^*)$ réalise l'inégalité $E_0(\tilde{x}^*, \tilde{\lambda}^*, \tilde{z}_u^*, \tilde{z}_l^*) \leq \epsilon_{\mathrm{tol}}$, le paramètre $\epsilon_{\mathrm{top}} > 0$ étant spécifié par l'utilisateur, avec une valeur par défaut égale à 10^{-8}.

Cette expression de l'erreur est également utilisée pour déterminer le critère d'arrêt des sous-problèmes associés aux différentes valeurs de μ. La solution approchée $(\tilde{x}_j^*, \tilde{\lambda}_j^*, \tilde{z}_{u,j}^*, \tilde{z}_{l,j}^*)$ pour le sous-problème $S(\mu_j)$ (2.19) doit vérifier :

$$E_{\mu_j}(\tilde{x}_j^*, \tilde{\lambda}_j^*, \tilde{z}_{u,j}^*, \tilde{z}_{l,j}^*) \leq \kappa_\epsilon \mu_j \tag{2.22}$$

κ_ϵ étant ici un paramètre d'ajustement dont la valeur par défaut à été choisie égale à 10 par les auteurs. L'algorithme reprend alors avec la valeur suivante du paramètre de barrière :

$$\mu_{j+1} = \max\left\{\frac{\epsilon_{\mathrm{tol}}}{10}, \min\left\{\kappa_\mu \mu_j, \mu_j^{\theta_\mu}\right\}\right\}$$

Ici encore, apparaissent de nouveaux paramètres d'ajustement de l'algorithme, κ_μ et θ_μ, que l'utilisateur peut choisir ou laisser à leurs valeurs par défaut (respectivement $0,2$ et $1,5$). L'utilisateur pourra également choisir d'autres stratégies pour la mise à jour du paramètre de barrière μ. On se réfèrera à la documentation en ligne figurant dans le site [58] pour de plus amples informations.

Résolution des sous-problèmes $S(\mu_k)$

La résolution de chacun des sous-problèmes 2.19 se fait comme dans l'algorithme 2. Le système associé à la recherche des directions de descente 2.20 subit cependant plusieurs modifications.

On procède tout d'abord à l'élimination des deux dernières lignes afin d'avoir à résoudre le système réduit suivant :

$$\begin{bmatrix} \nabla^2\mathcal{L}(x,\lambda) + \Sigma(x) & J_c(x)^{\mathrm{T}} \\ J_c(x) & 0 \end{bmatrix} \begin{pmatrix} d_x \\ d_\lambda \end{pmatrix} = -\begin{pmatrix} \nabla B(x,\mu) + J_c(x)^{\mathrm{T}}\lambda \\ c(x) \end{pmatrix} \tag{2.23}$$

système dans lequel B est la fonction de mérite 2.14 et $\Sigma(x) = (X_u - X)^{-1}Z_u + (X - X_l)^{-1}Z_l$. Les directions d_u et d_l associées aux multiplicateurs de Lagrange z_u et z_l se déduisent alors de d_x et d_λ par les relations :

$$d_u = (X_u - X)^{-1}Z_u d_x + z_u - (X_u - X)^{-1}\mu e$$
$$d_l = -(X - X_l)^{-1}Z_l d_x + z_l - (X - X_l)^{-1}\mu e$$

Avant la résolution du système 2.23, il est nécessaire de prendre quelques précautions afin de garantir de bonnes propriétés de convergence. D'après [93], la recherche linéaire employée nécessite que la projection dans le noyau de $J_c(x)$ du bloc supérieur gauche $\nabla^2\mathcal{L}(x,\lambda) + \Sigma(x)$ soit une matrice définie positive. Cette condition est vérifiée dès que la matrice complète possède exactement n valeurs propres strictement positives et m valeurs propres strictement négatives [77]. Ces valeurs propres sont donc dénombrées avant chaque calcul des directions de descente, et des matrices $\delta_w I_n$ et $-\delta_c I_m$ ($\delta_w, \delta_c \in \mathbb{R}^+$, adaptées automatiquement en fonction de μ, du rang de $J_c(x)$ et des signes des valeurs propres) sont respectivement ajoutées aux blocs supérieur gauche et inférieur droit, dans le cas où cette condition n'est pas vérifiée, afin de corriger cette défaillance. Cette phase, appelée *correction de l'inertie* est détaillée dans l'annexe consacrée à IPOPT.

Lorsque la matrice possède les bonnes propriétés, les directions d_x, d_λ, d_u et d_l sont calculées, puis un pas optimal est déterminé par la recherche linéaire décrite dans [93], et plus brièvement dans [92]. Nous donnons une courte description de cette méthode, ainsi qu'une rapide présentation de la notion de *filtre* dans la sous-section qui suit. Comme nous l'avons déjà indiqué, cette phase est essentielle pour assurer la convergence de l'algorithme pour n'importe quelle initialisation. Elle peut être désactivée lorsque l'on est sûr que l'initialiseur est suffisamment proche de l'optimum, auquel cas l'algorithme se comportera comme une méthode de Newton standard.

Le sous problème $S(\mu_k)$ est considéré comme résolu lorsque la condition 2.22 est remplie, ou que le nombre total d'itérations dépasse le maximum fixé par l'utilisateur. Dans le premier cas, l'algorithme reprend avec la nouvelle valeur du paramètre de barrière, en prenant l'optimum précédemment déterminé ainsi que les variables duales correspondantes pour point de départ.

Recherche linéaire - Filtre

A chaque itération de la résolution des sous-problèmes associés à une valeur du paramètre de barrière μ, lorsque les directions de descente $(d_x, d_\lambda, d_l, d_u)$ sont déterminées, la difficile phase de recherche linéaire sous contraintes doit être effectuée. Il s'agit de trouver une valeur de pas α telle que le nouvel itéré $(x_k + \alpha d_x, \lambda_k + \alpha d_\lambda, z_k^l + \alpha d_l, z_k^u + \alpha d_u)$ réalise une diminution acceptable de la fonction de mérite tout en vérifiant les contraintes. Cette complexe recherche linéaire sous contraintes est explicitée en annexe A.3. Une étude détaillée abordant notamment les questions de convergence figure dans l'article [93].

La méthode employée se base sur une réinterprétation du problème de minimisation 2.15 comme un problème de minimisation libre à deux objectifs pour lequel on cherche à minimiser simultanément $x \mapsto B(x,\mu)$ et $\theta : x \mapsto \|c(x)\|$, en insistant sur cette dernière fonction, par l'utilisation d'un *filtre*.

Le filtre est un ensemble $\mathcal{F} \subset \mathbb{R}^2$ initialisé par la donnée d'une valeur maximale de la mesure de violation des contraintes d'égalité $\theta_{\max}$:

$$\mathcal{F} = \left\{(\theta, \varphi) \in \mathbb{R}^2 \mid \theta \geq \theta_{\max}\right\}$$

A chaque fois que les directions de descente ont été déterminées et qu'une proposition de pas α est faite, associée à un itéré $x_{k,\alpha} = x_k + \alpha d_x$, cette proposition sera acceptée si le couple $(\theta_\alpha, \varphi_\alpha) = (\theta(x_{k,\alpha}), B(x_{k,\alpha}, \mu_k))$ n'appartient pas au filtre. Si le point est accepté, le filtre est augmenté par la relation :

$$\mathcal{F} \leftarrow \mathcal{F} \cup \left\{ (\theta, \varphi) \in \mathbb{R}^2 \mid \theta \geq (1 - \gamma_\theta)\theta(x_k) \text{ et } \varphi \geq B(x_k, \mu_k) - \gamma_\varphi \theta(x_k) \right\}$$

où γ_θ et γ_φ sont des constantes fixées au début de l'algorithme, à valeurs dans $]0, 1[$.

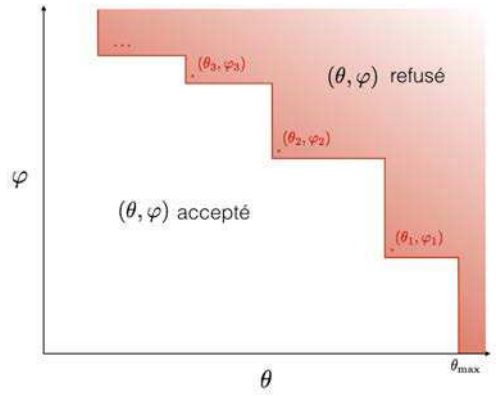


Figure 2.4: Test d'acceptabilité vis-à-vis du filtre

La figure 2.4 ci-contre illustre le test d'acceptabilité. Si le couple fonction coût / violation de la contrainte associé à la valeur du pas α appartient à la zone coloriée en rouge, alors la valeur du pas est refusée et un nouveau pas va être recherché, d'abord par tentative de correction de la direction de descente puis, si cette correction n'aboutit pas sur un pas satisfaisant, par poursuite de la procédure de recherche linéaire par réductions successives de la valeur du pas. Si malgré tout, on obtient des valeurs du pas trop petites, il ne sera plus possible d'aboutir à une réduction suffisante de B et θ et les points seront systématiquement refusés par le filtre. Dans ce cas, une seconde procédure de correction du point courant est amorcée, dont le but est de récupérer une valeur de x telle que $(\theta(x), B(x, \mu_k) \notin \mathcal{F}$, par minimisation de $\theta(x)$ par rapport à toutes les variables. Un échec de cette phase critique signifie l'arrêt complet de la procédure d'optimisation.

Cette stratégie de filtrage évite un certain nombre de problèmes soulevés par l'utilisation de recherche linéaire avec une fonction de mérite (voir [93]), comme la difficulté du choix d'un paramètre de pénalisation pour la contribution des contraintes dans la fonction de mérite, et d'autres liés à la convergence vers des points problématiques vis-à-vis de la vérification des contraintes. Enfin, cette méthode évite aussi l'apparition d'un cycle, dans lequel la recherche linéaire propose successivement les mêmes valeurs de pas qui réduisent tour à tour la fonction coût ou la mesure de violation des contraintes sans jamais vérifier les conditions de sélection d'un pas ([92]).

2.5.4 L'interface FreeFem++ pour IPOPT

Les différents prototypes pour la construction d'un problème d'optimisation et sa résolution par IPOPT depuis un script FreeFem++ peuvent être chargés avec le paquet `ff-Ipopt`. Comme nous l'avons vu, le problème le plus général que peut permettre de résoudre cette méthode est le suivant :

$$\begin{array}{ll} \text{Trouver} & x_0 = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} f(x) \\ \text{Tel que} & \left\{ \begin{array}{ll} \forall i \leq n,\ x_i^{\text{lb}} \leq x_i \leq x_i^{\text{ub}} & \text{(simples bornes)} \\ \forall i \leq m,\ c_i^{\text{lb}} \leq c_i(x) \leq c_i^{\text{ub}} & \text{(fonctions contraintes)} \end{array} \right. \end{array} \tag{2.24}$$

Dans ce formalisme, des contraintes d'égalité peuvent être introduites par un choix de bornes c_i^{lb} et c_i^{ub} identiques. De même, pour un indice i donné, une égalité entre x_i^{lb} et x_i^{ub}

impose une valeur à la composante correspondante de x ce qui, selon les options spécifiées à IPOPT, peut revenir à ne plus considérer x_i comme une variable.

Pour tirer pleinement avantage des fonctionnalités de IPOPT, d'après les développements théoriques précédents, il est nécessaire de fournir au logiciel une implémentation pour la fonction coût et les contraintes, mais aussi pour leurs dérivées premières, ainsi que pour la matrice hessienne des contraintes de la fonction $\mathcal{L}$ qui apparaît dans le système 2.23. L'appel le plus complet que l'on puisse stipuler sera donc le suivant :

```
7 int ipcode = IPOPT(f, df, d2L, c, Jc, x0,
8                    lb=xlb, ub=xub, clb=clb, cub=cub,
9                    ... /* paramètres optionnels */ ...);
```

Lorsque l'implémentation de l'une des dérivées du second ordre n'est pas disponible, l'algorithme n'en utilisera aucune et approchera l'inverse de la hessienne par une méthode de type BFGS à faible stockage.

Les prototypes des fonctions dont les identifiants apparaissent dans l'appel de cette routine doivent être les suivants :

```
1 func real f(real[int] &X) {/*implémentation de la fonction coût*/}
2 func real[int] df(real[int] &X) {/*gradient de la fonction coût*/}
3 func matrix d2L(real[int] &X,real objfactor,real[int] &lambda)
4 {/*implémentation de la hessienne du lagrangien*/}
5 func real[int] c(real[int] &X) {/*implémentation des contraintes*/}
6 func matrix Jc(real[int] &X) {/* jacobienne des contraintes*/}
```

On pourra s'étonner de voir apparaître une variable supplémentaire dans le prototype de l'implémentation de la hessienne du lagrangien en ligne 3 :

```
matrix (real[int] &,real,real[int] &)
```

La raison en est que les concepteurs d'IPOPT ont introduit un *facteur d'échelle*, sur lequel l'utilisateur peut d'ailleurs agir par l'intermédiaire du fichier d'options en spécifiant une valeur ou une méthode automatique de mise à l'échelle (se référer à [59] pour plus de précisions à propos de ce *scaling*). Ce facteur peut être amené à varier au cours de l'algorithme. Aussi, le lagrangien devient-il fonction de celui-ci, en plus de dépendre du point courant et des multiplicateurs de Lagrange : $\mathcal{L}(x,\sigma,\lambda) = \sigma f(x) + \sum_{i=1}^{m} \lambda_i c_i(x)$. La hessienne doit refléter cette dépendance et doit donc implémenter l'application suivante :

$$(x,\sigma,\lambda) \in \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^m \longmapsto \sigma\nabla^2 f + \sum_{i=1}^{m} \nabla^2 c_i(x)\lambda_i$$

Afin de contribuer à l'allègement du temps de développement recherché par la philosophie de FreeFem++, un effort de simplification pour l'utilisation de IPOPT par l'intermédiaire du logiciel a été réalisé par une surcharge intensive de la fonction appelant IPOPT. Il en résulte la possibilité d'omettre certaines des fonctions du problème d'optimisation lorsque par exemple celui-ci ne comporte pas de contraintes, ou encore de simplifier le prototype de la fonction implémentant la hessienne du lagrangien lorsque les contraintes sont linéaires, de ne passer que les couples matrice-vecteur caractérisant les problèmes quadratiques ou à contraintes linéaires, *etc.*

Enfin, pour ce qui concerne l'ensemble des options d'IPOPT, dont l'ajustement se fait à l'aide d'un fichier d'option dans l'interface C++ de IPOPT, le choix qui a été fait est de rendre directement accessibles via le script FreeFem les paramètres incontournables que l'on retrouve dans toutes les méthodes pratiques d'optimisation tels que les critères

d'arrêts, divers paramètres de tolérance et de pénalisation, *etc.* La totalité de ces paramètres d'ajustement reste accessible via le fichier d'options IPOPT natif. Les précisions les plus techniques en rapport avec cette interface sont compilées dans la documentation de FreeFem++ [79].

2.6 Optimisation stochastique

L'optimisation *stochastique* regroupe un ensemble de méthodes dans lesquelles la recherche des optima est basée sur l'évaluation de la fonctionnelle en un vaste ensemble de points appelé population que l'on fait évoluer selon des lois probabilistes. L'analyse stochastique et la notion de variable aléatoire y jouent donc un rôle central. L'introduction de l'aléatoire dans les processus d'optimisation présente de nombreux avantages. Il s'agit tout d'abord de techniques qui n'exploitent pas les dérivées de la fonction coût, ce qui permet donc de pouvoir optimiser des fonctions non dérivables, ou dont on ne saurait pas calculer les dérivées, comme c'est par exemple le cas lorsque l'évaluation de la fonction coût se base sur des logiciels propriétaires dont le code source n'est pas accessible et dont on ne sait rien de l'implémentation. L'approche probabiliste rend également possible l'optimisation de fonctions bruitées, ce qui n'est pas possible avec les algorithmes que nous avons abordés précédemment, qui comme nous l'avons vu, nécessitent un minimum de régularité aux fonctions impliquées. Un autre avantage certain de ces méthodes réside dans leur aptitude à converger *presque sûrement* vers le minimum global de la fonction coût, et ce le plus souvent sans que ne se pose le problème de l'initialisation de l'algorithme. Enfin, certains de ces algorithmes peuvent facilement s'appliquer à des problèmes d'optimisation de tous types, mélangeant optimisation continue et combinatoire. On parle alors de *métaheuristique*.

Ces remarquables propriétés ne sont néanmoins pas sans conséquences, et de tels avantages ne suffisent pas à occulter les nombreuses difficultés soulevées par leur utilisation. Ces algorithmes présentent en effet des propriétés de convergence assez médiocres, et nécessitent un grand nombre d'évaluations de la fonction coût. Quand on sait qu'une application industrielle modeste, peut requérir plusieurs heures de calcul pour une simple évaluation, on conçoit aisément qu'il n'est pas réaliste de procéder à une dizaine de ces évaluations pour chacune des milliers d'itérations nécessaires à la convergence de la méthode. Il est alors inévitable d'utiliser des modèles plus grossiers, quitte à les raffiner à mesure que l'on approche de la solution. De plus, conséquence de la difficulté d'établir une caractérisation des optima des fonctions non-dérivables (ou pire, non lipschitziennes) aisément traduisible numériquement, la formulation de critères d'arrêt pertinents pour ces méthodes est délicate et l'on doit bien souvent se contenter du nombre de *générations* (équivalent de l'itération dans les algorithmes classiques) ou du temps de calcul.

Ainsi, ces recherches aléatoires ne sont utilisées qu'en dernier recours lorsque les méthodes classiques échouent ou que la fonction coût ou son implémentation exclut toute autre alternative. Nous pensions initialement développer plusieurs méthodes de ce type pour FreeFem++ car celles-ci sont naturellement parallélisables et, à cette époque, la version MPI du logiciel venait d'être développée et le parallélisme avait alors le vent en poupe au sein de l'équipe FreeFem++. Cet objectif a par la suite été reconsidéré, avant tout en raison des performances modestes de ces algorithmes. Nous avons implémenté un premier algorithme génétique, très basique, à partir de la bibliothèque Evolving Objects ([39], [61]), une bibliothèque d'algorithmes d'optimisation stochastique écrite en C++ et aujourd'hui disponible par le biais de la version plus générale ParadiseEO [81]. Cet algorithme sans réelle prétention numérique avait essentiellement vocation à être utilisé en

tant qu'illustration pour un tutoriel sur l'écriture d'interfaces entre FreeFem++ et des logiciels ou bibliothèques tierce-partie pour les *Journées FreeFem++*. Enfin, nous avons introduit la possibilité d'utiliser un algorithme évolutif, développé par Nikolaus Hansen du Laboratoire de Recherche en Informatique à Orsay, et qui implémente une méthode CMA-ES (pour *covariance matrix adaptation evolution strategy*, [47]). Une version MPI de cette méthode est bien entendu disponible dans FreeFem++.

Les deux méthodes pré-citées ne sont *a priori* pas adaptées à des problèmes avec contraintes, exception faite de l'algorithme génétique qui peut naturellement imposer de simples bornes sur les paramètres. Les problèmes d'optimisation sous contraintes dans lesquels la fonction coût peut être définie lorsque les contraintes ne sont pas vérifiées peuvent néanmoins être résolus par minimisation d'une fonction de mérite pénalisant les contraintes comme nous l'avons vu au paragraphe 2.1. FreeFem++ ne propose pas à ce jour d'outil d'optimisation stochastique pour les problèmes dans lesquels la vérification de la contrainte est un impératif.

Pour les paragraphes qui suivent, nous supposerons que l'on tente d'optimiser une fonction $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ avec très peu d'hypothèses sur la régularité de f qui n'a pas nécessairement besoin d'être continue.

2.6.1 Les algorithmes génétiques

Dans le cadre des algorithmes génétiques, les éléments de la population sont appelés des *individus* et appartiennent ici à $\mathbb{R}^n$. L'algorithme modifie itérativement cette population en tentant de mimer ce qui se produit dans le monde du vivant. On procède pour cela à des *sélections* et des *croisements* entre individus, inspirés de l'évolution du patrimoine génétique des espèces. L'adaptabilité d'un individu se mesure par le biais de la fonction coût f (pour un problème de minimisation, les individus les plus adaptés sont ceux dont l'image par f est la plus petite). Comme dans la nature, un phénomène de *mutations* introduit de la nouveauté dans le patrimoine génétique de la population et permet d'espérer contrecarrer la stagnation dans un minimum local d'adaptabilité.

La phase d'évaluation concentre en général la plus grande partie de la complexité calculatoire de ces algorithmes. Il s'agit d'une boucle de calcul sans aucune dépendance des données (en dehors d'éventuelles dépendances internes à la fonction coût bien sûr), donc parallélisable de manière triviale avec une réduction du temps de calcul linéaire pour peu que le nombre de processeurs alloués soit inférieur à la taille de la population. Si l'on dispose d'une grande quantité de processeurs, on peut également augmenter la taille de la population sans incidence sur le temps de calcul, une population plus étendue permettant d'affermir les capacités de l'algorithme à déterminer un optimum global.

2.6.2 L'algorithme évolutif par adaptation de la matrice de covariance (CMA-ES)

Les *stratégies d'évolution*, en abrégé *ES* (tiré de leur appellation anglo-saxonne *evolution strategies*) constituent une sous-classe d'algorithmes génétiques. Pour une population de μ individus avec génération de λ enfants à chaque itération, on note une stratégie d'évolution dans laquelle chaque enfant n'a qu'un seul parent (μ, λ)-ES ou $(\mu + \lambda)$-ES selon que la sélection se fait respectivement uniquement sur les enfants ou sur l'ensemble des parents et de leur progéniture. Lorsque la génération des enfants se fait par recombinaison de ρ parents, l'algorithme est noté $(\mu/\rho, \lambda)$ ou $(\mu/\rho + \lambda)$-ES selon la portée de la sélection. L'algorithme (1,1)-ES revient par exemple à optimiser par recherche aléatoire. Quant à CMA-ES, qui est interfacé dans FreeFem++, c'est une stratégie d'évolution de

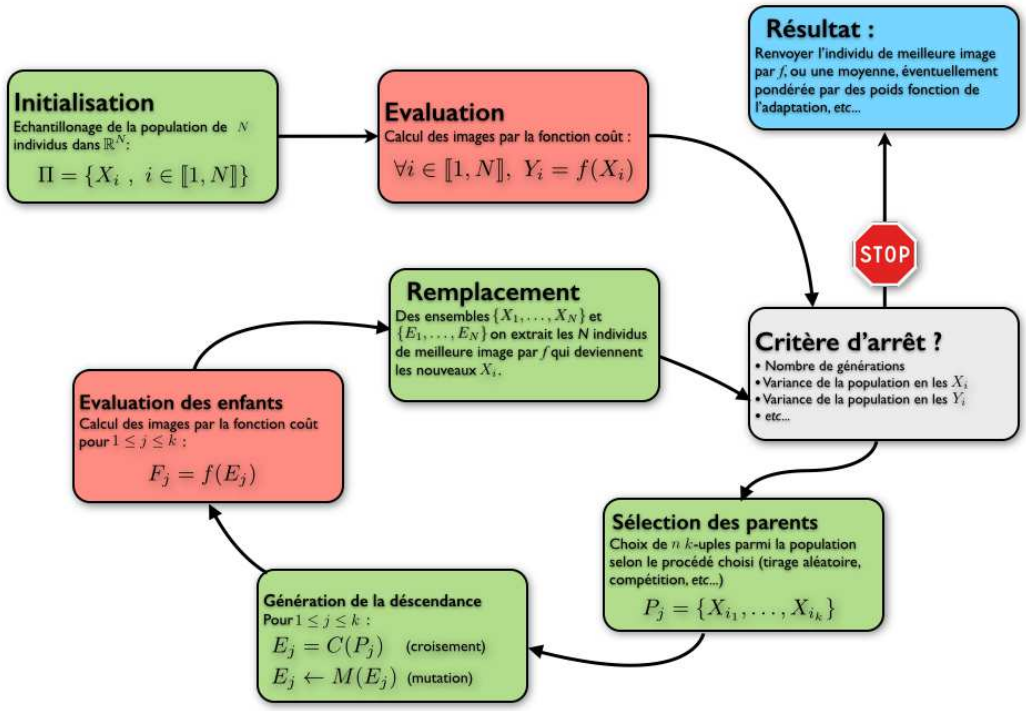


Figure 2.5: Schéma général d'un algorithme génétique

type $(\mu/\mu, \lambda)$ car les enfants y sont engendrés par une loi de probabilité conçue à partir de la totalité de la population.

Stratégie d'évolution : algorithme canonique

Une stratégie d'évolution, dans sa version la plus générale, se déroule selon le schéma algorithmique suivant : :

- Initialiser la population $E^{(0)} = \left\{x_i^{(0)} \in \mathbb{R}^n\ ,\ 1 \leq i \leq \mu\right\}$ et $k = 0$
- Tant que le critère d'arrêt n'est pas vérifié, faire :
  - Pour $i \in [\![1, \lambda]\!]$:
    Choisir une famille d'individus à ρ éléments, les parents : $(p_i)_{1 \leq i \leq \rho} \in {E^{(k)}}^{\rho}$
    Concevoir l'enfant e_i par *recombinaison* de la famille $(p_i)_{1 \leq i \leq \rho}$.
    Appliquer une *mutation* à l'enfant e_i
  - *Sélectionner* μ individus parmi $E^{(k)} \cup \{e_i\ ,\ 1 \leq i \leq \lambda\}$ (ou en se limitant aux seuls enfants $\{e_i\ ,\ 1 \leq i \leq \lambda\}$) qui constitueront la génération suivante $E^{(k+1)}$

Les notions de *recombinaison*, *mutation* et de *sélection* sont ici assez vagues. On peut conserver un processus de sélection aléatoire comme pour les algorithmes génétiques, ou bien pratiquer une sélection déterministe en ne gardant que les λ meilleurs individus. Dans certaines méthodes, comme celle que nous allons voir, la sélection des parents n'est pratiquée qu'une seule fois à chaque génération, les individus sélectionnés sont utilisés pour déterminer les caractéristiques d'une loi normale multidimensionnelle, selon laquelle les enfants seront échantillonnés. Ce procédé unifie recombinaison et mutation, et permet ainsi d'obtenir une convergence linéaire en adaptant la loi d'échantillonnage en fonction

des informations sur la fonction coût que l'on peut obtenir au travers de l'évaluation des images des individus.

Loi normale multidimensionnelle

La *loi normale multidimensionnelle*, ou *loi multivariée* généralise la notion de loi normale (gaussienne) au cas des vecteurs aléatoires. Toute loi multivariée en dimension n est entièrement définie par la donnée d'un vecteur $\mathbf{m} \in \mathbb{R}^n$, représentant la moyenne du vecteur aléatoire, et d'une matrice définie positive $C \in \mathcal{M}_n(\mathbb{R})$. On la note alors $\mathcal{N}(\mathbf{m}, C)$ et sa *densité de probabilité* s'écrit :

$$p_{\mathcal{N}(\mathbf{m},C)}(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}\sqrt{\det C}} e^{\frac{1}{2}(\mathbf{x}-\mathbf{m})^{\mathrm{T}} C^{-1}(\mathbf{x}-\mathbf{m})} \tag{2.25}$$

La loi normale multivariée peut par ailleurs s'exprimer en fonction de la loi multidimensionnelle unitaire, associée à n variables aléatoires réelles indépendantes de même loi Gaussienne centrée réduite $\mathcal{N}(\mathbf{0}, I)$:

$$\mathcal{N}(\mathbf{m}, C) = \mathbf{m} + \mathcal{N}(\mathbf{0}, C) = \mathbf{m} + BD\ B^{\mathrm{T}}\mathcal{N}(\mathbf{0}, I) = \mathbf{m} + BD\mathcal{N}(\mathbf{0}, I) \tag{2.26}$$

où B est la matrice des vecteurs propres de C et D la matrice diagonale des racines carrées des valeurs propres de C : $C = BD^2\ B^{\mathrm{T}}$.

D'un point de vue géométrique, les ellipsoïdes $\left\{\mathbf{x} \in \mathbb{R}^n \mid (\mathbf{x}-\mathbf{m})^{\mathrm{T}} C^{-1}(\mathbf{x}-\mathbf{m}) = k\right\}$, $k \in \mathbb{R}_+^*$, sont les isosurfaces de la densité de probabilité de $\mathcal{N}(\mathbf{m}, C)$ et, lors de l'échantillonnage de points selon cette loi, ceux-ci se répartissent pour la plupart à l'intérieur de l'ellipsoïde $k = 1/4$ au-delà duquel la probabilité d'échantillonner un point est quasiment nulle. Les vecteurs de B représentent les axes principaux de ces ellipsoïdes. Aussi, en choisissant une matrice bien adaptée, on peut jouer sur la direction de ces axes, ainsi que sur les étendues sur lesquelles les enfants seront échantillonnés le long de ces axes, afin d'obtenir des échantillons qui auront tendance à réduire la valeur de la fonction coût. Par exemple pour le cas d'une fonction convexe deux fois continûment dérivable, un choix localement optimal pour C est de prendre cette matrice égale à l'inverse de la hessienne au point considéré [47]. L'algorithme CMA-ES essaie donc de construire une sorte d'approximation stochastique de la hessienne à chaque itération afin de générer des échantillons bien adaptés à la minimisation de la fonction coût. Cette approximation est un objet statistique, aussi a-t-elle un sens dans le cas de fonctions non dérivables. Ainsi, à chaque itération k de l'algorithme sont déterminés une moyenne $\mathbf{m}^{(k)}$, correspondant à l'itéré courant, une matrice de covariance $C^{(k)}$ et un écart type global $\sigma^{(k)}$, qui permettent d'échantillonner les λ enfants par la loi $\mathcal{N}\left(\mathbf{m}^{(k)}, {\sigma^{(k)}}^2 C^{(k)}\right)$.

Sélection et recombinaison par la moyenne

Dans la stratégie d'évolution CMA, l'étape de sélection et de recombinaison est effectuée par le calcul du centre de la loi multivariée. Cette mise à jour se fait par une moyenne pondérée des μ individus possédant la plus petite image par la fonction coût. A l'itération k, lorsque la population courante $\mathbf{x}_1^{(k)}, \dots \mathbf{x}_\lambda^{(k)}$ a été échantillonnée, cette moyenne s'écrit :

$$\mathbf{m}^{(k)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(k)} \tag{2.27}$$

où les $\mathbf{x}_{1:\lambda}^{(k)}, \dots, \mathbf{x}_{\lambda:\lambda}^{(k)}$ sont les individus de la génération k ré-indexés de sorte à ce que $f\left(\mathbf{x}_{1:\lambda}^{(k)}\right) \leq f\left(\mathbf{x}_{2:\lambda}^{(k)}\right) \leq \cdots \leq f\left(\mathbf{x}_{\lambda:\lambda}^{(k)}\right)$ et les w_i désignent une famille décroissante de réels positifs.

Adaptation de la matrice de covariance

L'évolution de la population vers des individus réalisant des valeurs optimales de la fonction coût est assurée par la mise à jour de la matrice de covariance de la distribution, puisque c'est grâce à elle qu'à chaque itération la nouvelle population est échantillonnée. Dans la méthode CMA-ES, cette mise à jour est effectuée à l'aide d'une variante d'un estimateur non-biaisé du maximum de vraisemblance, calculé à partir des μ individus les plus aptes :

$$C^{(k+1)} = \sum_{i=1}^{\mu} w_i \left(\mathbf{x}_{i:\lambda}^{(k+1)} - \mathbf{m}^{(k)}\right) \left(\mathbf{x}_{i:\lambda}^{(k+1)} - \mathbf{m}^{(k)}\right)^{\mathrm{T}} \tag{2.28}$$

Cette matrice est ensuite modifiée selon divers schémas correctifs explicités en annexes et commentés dans [47].

2.6.3 L'interface FreeFem++ pour CMA-ES

La version initiale écrite en C de l'algorithme peut être récupérée sur le site [31]. L'interface est compilée dans le module `ff-cmaes`. Une interface pour FreeFem++-mpi a également été développée, dans laquelle l'évaluation de la population est calculée en parallèle sur les divers processeurs disponibles. C'est cette version parallélisée que nous avons utilisée pour la résolution du problème des trois ellipses présenté à Jyvaskyla (section 3.3). Il est d'ailleurs plus que conseillé de systématiquement utiliser cette version parallèle, même pour une utilisation sur un ordinateur personnel, puisque la plupart des machines contemporaines comportent des processeurs multi-cœurs dont il serait dommage de ne pas exploiter les avantages. Nous renvoyons encore une fois le lecteur à la documentation de FreeFem++ concernant le nom des routines et l'utilisation de celles-ci. Il est important de savoir que, ici aussi, l'implémentation utilise des matrices pleines de la taille du nombre de paramètres d'optimisation. Il n'est donc pas possible de résoudre des problèmes de grande taille avec cet optimiseur.

Chapitre 3

Applications

Nous présentons ici une série d'expérimentations numériques menées à l'aide des différents outils d'optimisation que nous avons implémentés pendant cette thèse et que nous avons présentés dans le chapitre qui précède. Nous commençons par une petite application informelle qui nous a servi de test systématique pour valider les nouvelles implémentations. Puis nous continuons avec quelques applications plus sérieuses, dont la simulation de condensats de Bose-Einstein par la minimisation de l'énergie de Gross-Pitaevsky. La dernière application consiste en la résolution d'un problème de contact avec conditions aux bords de Signorini. Les résultats obtenus sur ce dernier problème ont été exploités dans un article écrit avec Zakaria Belhachmi, Faker Ben Belgacem et Frédéric Hecht, que nous incluons dans son intégralité.

3.1 Considérations d'ordre général

Dans la plupart des problèmes rencontrés, on recherche une solution dans un espace fonctionnel qu'il faut discrétiser, afin de se ramener à une recherche de minimiseur dans un espace de dimension finie. Pour cela, on procède de manière classique en éléments finis par la construction d'une triangulation $\mathcal{T}_h$ du domaine Ω sur lequel sont définis les éléments de l'espace fonctionnel, puis par l'approximation de l'espace par les ensembles de fonctions polynomiales par morceaux sur chacun des éléments de la triangulation précédemment définie. On travaillera donc avec des éléments finis de Lagrange, le plus souvent P^1, en ayant parfois recours à des éléments de degré deux (éléments P^2) lorsque la situation l'exige.

On utilisera autant que faire se peut les outils d'adaptation de maillage implémentés dans FreeFem++. Les algorithmes d'optimisation travaillant en dimension fixe, le maillage ne saurait être adapté pendant l'exécution du processus de minimisation quand on optimise par rapport à la fonction éléments finis, puisque la modification du maillage va entraîner celle du nombre de degrés de liberté. Dans ces cas, une première minimisation est donc effectuée, puis interrompue lorsqu'un premier niveau de précision a été atteint. Le maillage est adapté à cette première solution qui est alors interpolée sur le nouveau maillage, avant d'être utilisée pour initialiser un nouveau processus d'optimisation. Ce procédé peut être répété à volonté, mais en général, trois itérations sont suffisantes. Selon les algorithmes, il est plus ou moins possible de récupérer les valeurs des variables internes et de les restituer au nouvel appel de la routine d'optimisation, avec adaptation à la nouvelle géométrie pour celles qui en dépendent, et pour lesquelles on est capable d'anticiper le mécanisme de cette dépendance. Ceci permet de ne pas naïvement recommencer tout l'algorithme en ne modifiant que le nombre de variables et l'initialiseur. On peut alors réaliser l'économie

d'un nombre considérable d'itérations.

3.2 Surfaces minimales

Au-delà des problèmes très simples de minimisation de fonctionnelles quadratiques associées à la résolution des équations de Laplace ou de Stokes, l'approximation numérique de surfaces minimales est un test très pertinent pour mettre à l'épreuve et valider de nouveaux algorithmes. Ces problèmes présentent de fortes non linéarités tout en restant peu coûteux du point de vue de l'implémentation, puisque les expressions des dérivées prennent des formes simples et que celles-ci sont directement transcriptibles dans l'algèbre de fonctions de FreeFem++. On dispose ainsi de problèmes à la fois difficiles d'un point de vue numérique et économiques pour ce qui est du développement.

3.2.1 Cadre général

De manière générale, un problème de surface minimale consiste à, étant donné un ensemble $\mathcal{S}$ de sous-variétés de $\mathbb{R}^3$, trouver $\underset{S\in\mathcal{S}}{\operatorname{argmin}}\ \mathcal{A}(S)$, où $\mathcal{A}$ est l'*aire* de la surface S. Dans la pratique, il faut bien entendu préciser $\mathcal{S}$ et, afin d'éviter toute difficulté dans le sens à donner à $\mathcal{A}$, il est naturel de se limiter au cas où $\mathcal{S}$ est un ensemble de nappes paramétrées (dans un système de coordonnées arbitraire), avec un paramétrage de régularité convenable. Afin de simplifier l'exposé, nous assimilerons surface et paramétrage (même si une même surface peut avoir plusieurs paramétrages distincts). On se donne Ω un ouvert borné et connexe de $\mathbb{R}^2$, et on dira qu'une surface paramétrée sur Ω est la donnée d'une application de $\overline{\Omega}$ dans $\mathbb{R}^3$, de classe C^2 dans Ω, et de restriction à $\partial\Omega$ de classe C^1 :

$$\mathcal{S}(\Omega) = \left\{ X \in \mathcal{F}(\overline{\Omega}, \mathbb{R}^3) \mid X|_\Omega \in C^2(\Omega, \mathbb{R}^3) \text{ et } X|_{\partial\Omega} \in C^1(\partial\Omega, \mathbb{R}^3) \right\} \tag{3.1}$$

Ainsi, l'application A qui à une surface associe son aire devient une fonctionnelle sur $\mathcal{S}(\Omega)$ que nous pouvons facilement expliciter en fonction du paramétrage, pour peu que l'on dispose d'un bon manuel de géométrie différentielle :

$$\mathcal{A}(X) = \int_\Omega \sqrt{g} = \int_\Omega \|\partial_u X \wedge \partial_v X\| \tag{3.2}$$

où g est le déterminant de la *première forme fondamentale* associée à X, dont les coefficients de l'expression matricielle dans le système de coordonnées choisi sont les produits scalaires $g_{ij} = \partial_i X.\partial_j X$.

La minimisation de la fonctionnelle 3.2 est un problème mal posé puisque pour tout difféomorphisme φ de Ω dans lui-même, on a $(\mathcal{A} \circ \varphi)(X) = \mathcal{A}(X),\ \forall X$. Pour espérer obtenir un résultat dans toutes les situations, il faudrait envisager ce problème d'optimisation sous un autre angle. Par exemple en choisissant de minimiser une fonctionnelle plus simple, convexe ou au moins coercive, pour laquelle la stationnarité de la fonctionnelle d'aire serait une contrainte. Cela compliquerait néanmoins beaucoup le problème. Nous allons donc nous limiter au cas plus simple des surfaces qui sont les graphes d'applications de Ω dans $\mathbb{R}$. En notant :

$$S(\Omega) = \left\{ f : \overline{\Omega} \to \mathbb{R} \;\middle|\; f|_\Omega \in C^2(\Omega, \mathbb{R}) \text{ et } f|_{\partial\Omega} \in C^1(\partial\Omega, \mathbb{R}) \right\},$$

on obtient l'expression de la fonctionnelle d'aire en injectant le paramétrage $X(u, v) = (u, v, f(u, v))$ dans 3.2, ce qui définit la fonctionnelle sur $S(\Omega)$ suivante :

$$A(f) = \int_\Omega \sqrt{1 + |\nabla f|^2} \tag{3.3}$$

Nous allons alors aborder le problème suivant, dit *de Plateau*[1] :
Problème de Plateau : *Trouver la surface d'aire minimale s'appuyant sur une courbe donnée. C'est-à-dire, étant donnée une fonction* $\Gamma \in C^1(\partial\Omega, \mathbb{R})$*, trouver :*

$$u^* = \underset{u \in S(\Omega)}{\operatorname{argmin}} \; A(u) \quad avec \;\; u^*|_{\partial\Omega} = \Gamma \tag{3.4}$$

On tente de résoudre ces problèmes par une recherche de points critiques pour les fonctionnelles $\mathcal{A}$ et A. Il s'agit donc de trouver des fonctions u dans $S(\Omega)$ (resp. dans $\mathcal{S}(\Omega)$) annulant la différentielle de A (resp. $\mathcal{A}$) au sens de Fréchet $dA(u)$. En notant $S_0(\Omega)$ (resp. $\mathcal{S}_0(\Omega)$) l'ensemble des fonctions de $S(\Omega)$ (resp. de $\mathcal{S}(\Omega)$) s'annulant sur $\partial\Omega$, ces dérivées s'écrivent :

$$\forall v \in S_0(\Omega) \; dA(u)(v) = \int_\Omega \left(1 + |\nabla u|^2\right)^{-1/2} \nabla u.\nabla v \tag{3.5}$$

Et la seconde variation, $\forall v, w \in S_0(\Omega)$:

$$d^2A(u)(v,w) = \int_\Omega \left(1 + |\nabla u|^2\right)^{-1/2} \nabla w.\nabla v - \int_\Omega \left(1 + |\nabla u|^2\right)^{-3/2} (\nabla u.\nabla w)(\nabla u.\nabla v) \tag{3.6}$$

Calcul des variations, solutions analytiques

Dans le cas où l'on se limite aux graphes de fonctions de $\mathbb{R}^2$ dans $\mathbb{R}$, on peut déterminer l'équation différentielle que doit vérifier cette fonction en écrivant l'équation d'Euler-Lagrange associée à la minimisation de 3.3 :

$$\frac{\partial}{\partial x}\left(\frac{\dfrac{\partial f}{\partial x}}{\sqrt{1+|\nabla f|^2}}\right) + \frac{\partial}{\partial y}\left(\frac{\dfrac{\partial f}{\partial y}}{\sqrt{1+|\nabla f|^2}}\right) \;=\; 0 \tag{3.7}$$

Relevons la similitude de cette expression avec celle de la courbure moyenne de la surface :

$$2H = -\nabla.n \tag{3.8}$$

Le vecteur normal à la surface se calculant comme :

$$n = \frac{\dfrac{\partial F}{\partial x} \wedge \dfrac{\partial F}{\partial y}}{\left\| \dfrac{\partial F}{\partial x} \wedge \dfrac{\partial F}{\partial y} \right\|} = \frac{1}{\sqrt{1+|\nabla f|^2}} \begin{pmatrix} -\partial_x f \\ -\partial_y f \\ 1 \end{pmatrix} \tag{3.9}$$

L'équation 3.7 équivaut donc à l'annulation de H et les surfaces d'aire minimale sont celles dont la courbure moyenne est nulle. Cette équation peut être remaniée pour obtenir la forme épurée, mais toujours fortement non linéaire suivante :

$$\frac{\partial^2 f}{\partial x^2}\left[1 + \left(\frac{\partial f}{\partial y}\right)^2\right] - 2\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\frac{\partial^2 f}{\partial x \partial y} + \frac{\partial^2 f}{\partial y^2}\left[1 + \left(\frac{\partial f}{\partial x}\right)^2\right] \;=\; 0 \tag{3.10}$$

1. **Joseph Plateau** (1801 -1883) : physicien et mathématicien belge célèbre pour ses travaux sur la persistance rétinienne et la synthèse du mouvement. Ses résultats sur les surfaces minimales sont le fruit de recherches sur les tensions superficielles et les phénomènes de capillarité.

Il peut être intéressant de remarquer que l'annulation de 3.5 pour toute fonction de S s'annulant sur le bord constitue la formulation variationnelle de l'équation ci-dessus. Cette formulation faible est difficile à obtenir en partant de l'équation 3.10.

Cette équation permet de déterminer quelques solutions analytiques à la main. Des mathématiciens ont ainsi pu dériver des solutions en recherchant des paramétrages possédant certaines particularités. En recherchant par exemple une fonction de la forme $f(x,y) = g(x) + h(y)$, la solution est le paramétrage d'une surface de Scherk [2], dont l'équation implicite est $e^z \cos(y) = \cos(x)$, et correspond donc au graphe de la fonction :

$$f(x,y) = \ln(\cos x) - \ln(\cos y) \ , \ x, y \not\equiv \frac{\pi}{2}[\pi] \qquad (3.11)$$

Des solutions de la forme $f(x,y) = g(x)/h(y)$ permettent d'identifier l'*hélicoïde* en tant que surface minimale s'appuyant sur une hélice. Une hélicoïde de rayon R effectuant n tours peut se paramétrer par :

$$\begin{cases} x = v\cos(u) \\ y = v\sin(u) \\ z = u \end{cases} \quad (u,v) \in [0, 2n\pi] \times [0, R] \qquad (3.12)$$

FIGURE 3.1: Surface de Scherk

Une dernière solution analytique est la *caténoïde*, dont la forme est celle d'un film de savon rejoignant deux cercles coaxiaux. Le cas de cercles de rayons unité, de centres $(0,0,-1)$ et $(0,0,1)$ admet pour paramétrage :

$$\begin{cases} x = \cosh(u)\cos(v) \\ y = \cosh(u)\sin(v) \\ z = u \end{cases} \quad (u,v) \in [-1,1] \times [0, 2\pi] \qquad (3.13)$$

3.2.2 Résolution numérique

Les problèmes sont discrétisés de manière classique en éléments finis. Si on appelle $\mathcal{T}_h$ une triangulation de Ω de précision h et comportant n points, on note V_h l'espace des fonctions P^1 sur chacun des éléments de $\mathcal{T}_h$ et $(\varphi_i)_{1\leq i\leq n}$ les fonctions de bases de V_h. A une fonction u de V_h, on associe le vecteur $U = (u_i)_{1\leq i\leq n} \in \mathbb{R}^n$ des composantes de u dans la base proposée, de sorte à ce que $u = \sum_{i=1}^n u_i\varphi_i$.

On peut tout d'abord essayer de résoudre 3.10 par une méthode de Newton appliquée à la formulation variationnelle :

$$\textit{trouver } u \in H^1(\Omega) \textit{ telle que } u|_{\partial\Omega} = \Gamma \textit{ et } \forall v \in H_0^1(\Omega), \ \int_\Omega \frac{\nabla v.\nabla u}{\sqrt{1+|\nabla u|^2}} = 0 \qquad (3.14)$$

On initialise pour cela u_0 avec une extension de Γ à tout Ω puis, pour tout $k \in \mathbb{N}$, on construit $u_{k+1} = u_k - h_k$ où h_k est la fonction de $H_0^1(\Omega)$ solution de :

$$\forall v \in H_0^1(\Omega), \ \int_\Omega \frac{\nabla h_k.\nabla v}{\sqrt{1+|\nabla u_k|^2}} - \int_\Omega \frac{(\nabla u_k.\nabla h_k)(\nabla u_k.\nabla v)}{(1+|\nabla u_k|^2)^{3/2}} = \int_\Omega \frac{\nabla u_k.\nabla v}{\sqrt{1+|\nabla u_k|^2}}$$

2. **Heinrich Ferdinand Scherk** (1798 -1885) : mathématicien allemand ayant publié des travaux sur les surfaces minimales et les nombres premiers

Mais la stabilité de la méthode serait trop dépendante de son initialisation. Beaucoup de cas nécessiteraient une première phase de minimisation à l'aide d'un algorithme plus stable afin de s'approcher suffisamment de la surface optimale. Nous allons donc utiliser les outils que nous avons présentés dans le chapitre précédent.

La transcription de la fonctionnelle dans un script FreeFem++ est immédiate. On suppose que l'on dispose d'une triangulation identifiée par `Th` et de la macro `Grad` décrite dans le premier chapitre. L'implémentation ne présente pas de difficulté particulière :

```
1 func real A(real[int] &U)
2 {
3   Vh u;
4   u[] = U;//fabrique une fonction EF de composantes U
5   real area = int2d(Th)(sqrt(1 + Grad(u)'*Grad(u)));
6   return area;
7 }
```

Pour ce qui est des dérivées, lorsque les restrictions à V_h de 3.5 et 3.6 sont évaluées en u, on obtient respectivement un vecteur et une matrice dont les composantes dans la base $(\varphi_i)_{1\leq i\leq n}$ sont :

$$\frac{\partial A}{\partial u_i}(u) = \int_{\mathcal{T}_h} \frac{\nabla u.\nabla\varphi_i}{\sqrt{1+|\nabla u|^2}} \tag{3.15}$$

$$\frac{\partial^2 A}{\partial u_i \partial u_j}(u) = \int_{\mathcal{T}_h} \left(\frac{\nabla\varphi_i.\nabla\varphi_j}{\sqrt{1+|\nabla u|^2}} - \frac{(\nabla u.\nabla\varphi_i)\,(\nabla u.\nabla\varphi_j)}{(1+|\nabla u|^2)^{3/2}} \right) \tag{3.16}$$

Une approche naïve consisterait à construire le gradient composante par composante en calculant à chaque fois 3.15. L'assemblage des matrices hessiennes se montrerait quant à lui plus délicat puisqu'il faudrait d'abord déterminer sa structure avant de calculer chaque coefficient par la formule 3.16. La manière la plus élégante est d'utiliser la commande `varf` qui va réaliser à elle seule toutes ces étapes.

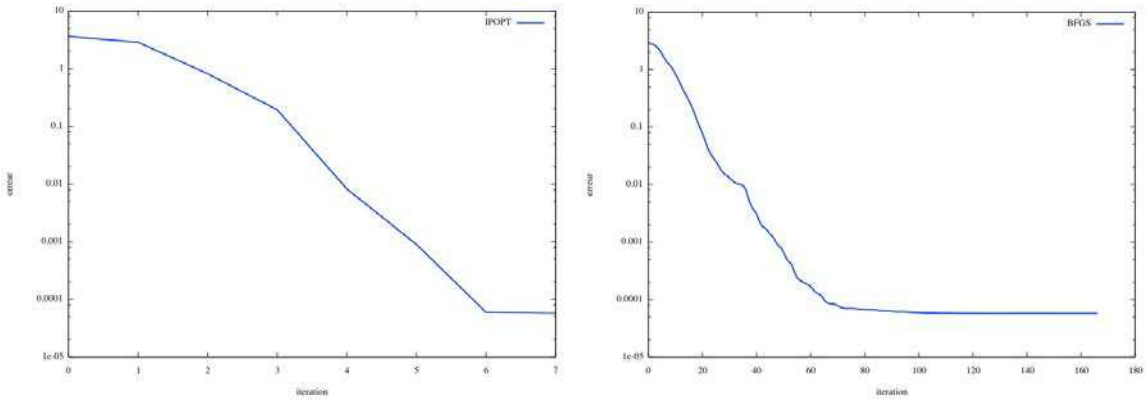
```
8  func real[int] dA(real[int] &U)
9  {
10   Vh u; u[]=U;
11   varf vfdA(w,v) =
12     int2d(Th)( 1./sqrt(1 + Grad(u)'*Grad(u)) * Grad(u)'*Grad(v) );
13   real[int] grad = vfdA(0,Vh);
14   return grad;
15 }
16
17 matrix hessian;//nécessite une variable globale
18 func matrix d2A(real[int] &U)
19 {
20   Vh u; u[]=U;
21   Vh alpha = 1./sqrt(1 + Grad(u)'*Grad(u)),
22      beta = 1./sqrt(1+Grad(u)'*Grad(u))^3;
23   varf vfd2A(w,v) =
24     int2d(Th)(
25         1./sqrt(1+Grad(u)'*Grad(u)) * Grad(w)'*Grad(v)
26       - 1./(sqrt(1+Grad(u)'*Grad(u))^3)
27           *(Grad(u)'*Grad(w))*(Grad(u)'*Grad(v)));
```

```
28   hessian = vfd2A(Vh,Vh);
29   return hessian;
30 }
```

Implémentées de cette façon, ces fonctions peuvent être utilisées avec n'importe laquelle des routines d'optimisation interfacées dans FreeFem++, à la condition que celle-ci prenne en compte les contraintes de bornes simples. Pour les routines ne supportant aucune contrainte, il est nécessaire de modifier l'implémentation de `dA` pour annuler les composantes du gradient sur le bord :

```
//si le bord a pour label 1:
varf vfdA(w,v) =
    int2d(Th)( 1./sqrt(1+Grad(u)'*Grad(u)) * Grad(u)'*Grad(v) )
  + on(1,w=0):
real[int] grad = vfdA(0,Vh);
```

L'implémentation des dérivées secondes devrait subir le même traitement, mais ce n'est pas réellement utile, car aucun des optimiseurs nécessitant ces modifications n'utilise la hessienne. L'élément le plus discriminant pour les routines utilisées demeure la précision de la triangulation. Seul IPOPT pourra en effet être utilisé si l'on désire utiliser un maillage très fin. Cela dit, les autres algorithmes montrent des performances très médiocres comparés à IPOPT avec des convergences en centaines d'itérations pour les méthodes de type quasi-Newton, et en dizaines de milliers pour la méthode stochastique CMA-ES, alors que la méthode de points intérieurs converge en une dizaine d'itérations.



(a) IPOPT **(b)** BFGS

FIGURE 3.2: Erreur en norme L^2 commise sur le minimiseur en fonction des itérations pour les méthodes IPOPT et BFGS

Les courbes 3.2 représentent la norme L^2 de la différence entre la fonction obtenue et la solution exacte, dans le cas où l'on essaie de retrouver la surface de Scherk (figure 3.1), en se limitant à une partie susceptible d'être représentée par le graphe d'une fonction bornée d'un ouvert de $\Omega \subset \mathbb{R}^2$ dans $\mathbb{R}$ (par exemple $\Omega =]-\frac{\pi}{2}+\epsilon, \frac{\pi}{2}+\epsilon[^2$). Les deux méthodes, IPOPT et BFGS l'approchent avec la même précision. Cependant la première le fait en sept itérations, alors que la seconde en nécessite environ cent soixante.

On peut ensuite s'amuser à résoudre le problème avec des contours moins orthodoxes et observer les surfaces minimales que l'on obtiendrait. Les surfaces représentées dans la figure 3.3 en sont quelques exemples.

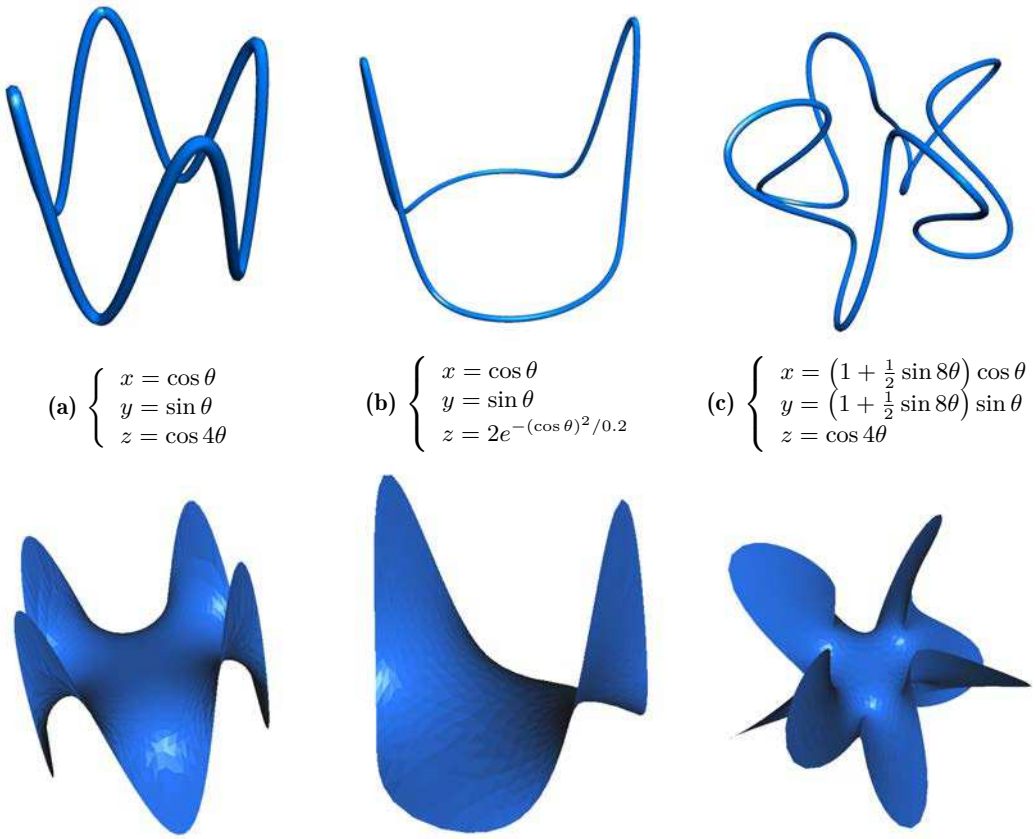


Figure 3.3: Surfaces minimales s'appuyant sur différents contours donnés par leur paramétrage cartésien ($\theta \in [0, 2\pi]$)

3.3 Un problème géométrique inverse

Dans le cadre du *Database Workshop for multiphysics optimization software* qui a eu lieu pendant l'hiver 2009-2010 à l'université de Jyväskylä en Finlande, ainsi qu'à Las Palmas en Espagne l'année suivante, nous avons traité un problème géométrique inverse en collaboration avec Jacques Périaux et Jyri Leskinen. Mêlant optimisation de forme et mécanique des fluides, ce problème nous a permis l'étude numérique de l'effet d'une utilisation intensive de l'adaptation de maillage sur la régularité de la fonctionnelle considérée. En particulier, nous avons observé l'apparition d'un *bruit* potentiellement fatal pour les méthodes d'optimisation nécessitant de la régularité à la fonctionnelle lorsque celle-ci est définie par une intégrale de bord faisant intervenir le champ de pression issu du problème de Navier-Stokes ou Stokes (mais dans une moindre mesure puisque le maillage sera moins affecté dans ce cas ; on peut d'ailleurs faire l'économie de l'adaptation de maillage pour ce modèle). Les variations peuvent être telles que même les méthodes les moins exigeantes en termes de régularité peuvent échouer à déterminer un minimiseur avec précision. L'alternative consistant à ne pas adapter le maillage n'étant pas une option, nous avons opté pour une approche tentant de réduire les variations aléatoires de la fonctionnelle causées par l'adaptation du maillage, avec un certain succès.

3.3.1 Définition du problème

On considère le domaine de base rectangulaire suivant :

$$\Omega_0 = \left\{(x,y) \in \mathbb{R}^2 \mid -\frac{L}{2} < x < \frac{L}{2} \text{ et } -\frac{l}{2} < y < \frac{l}{2}\right\}$$

Pour $\mathbf{x} \in \mathbb{R}^2$, on note $\tau_{\mathbf{x}}$ la translation de vecteur $\mathbf{x}$, et pour $\theta \in \mathbb{R}$, R_θ la rotation de centre O et d'angle θ. On considère les trois ellipses de référence suivantes :

$$\mathcal{E}_1^0 = R_{\theta_1^0}\left(\left\{(x,y) \in \mathbb{R}^2 \mid \left(\frac{x-x_1^0}{A}\right)^2 + \left(\frac{y-y_1^0}{a}\right)^2 \leq 1\right\}\right)$$
$$\mathcal{E}_0 = \left\{(x,y) \in \mathbb{R}^2 \mid \left(\frac{x}{B}\right)^2 + \left(\frac{y}{b}\right)^2 \leq 1\right\}$$
$$\mathcal{E}_2^0 = R_{\theta_2^0}\left(\left\{(x,y) \in \mathbb{R}^2 \mid \left(\frac{x-x_2^0}{A}\right)^2 + \left(\frac{y-y_2^0}{a}\right)^2 \leq 1\right\}\right)$$

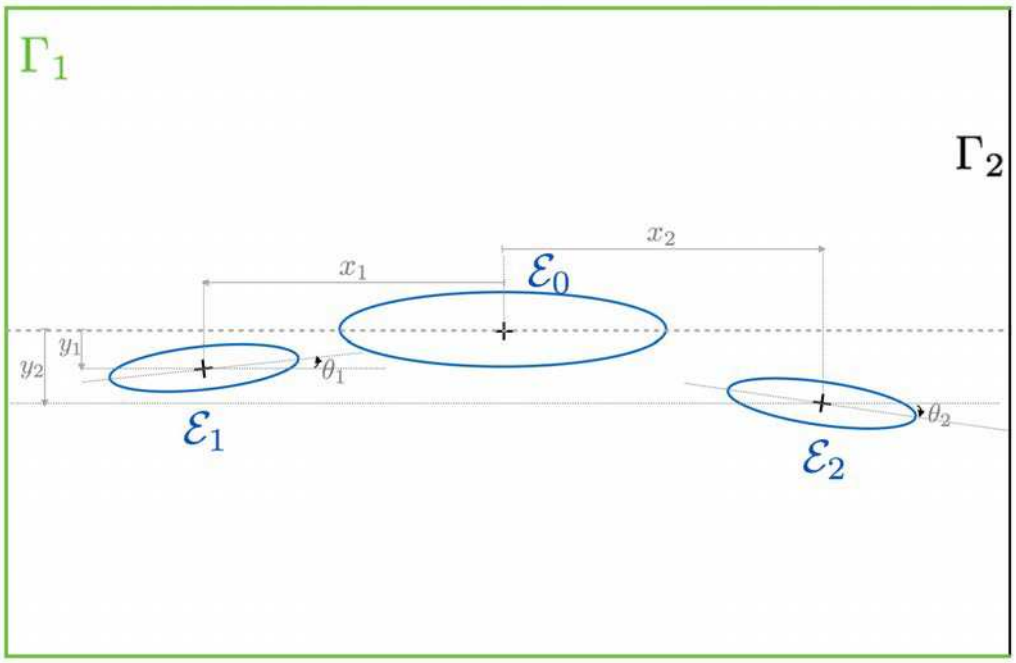


FIGURE 3.4: Domaine de calcul $\Omega(x_1, y_1, \theta_1, x_2, y_2, \theta_2)$ (les proportions ne sont pas respectées)

On définit alors les applications suivantes :

$$\begin{array}{rcl} \mathcal{E}_i : \mathbb{R}^2 \times \mathbb{R} & \longrightarrow & \mathcal{P}\left(\mathbb{R}^2\right) \\ (\mathbf{x}, \theta) & \longmapsto & \left(\tau_{\mathbf{x}} \circ R_{\theta - \theta_i^0} \circ \tau_{-\mathbf{x}_i^0}\right)(\mathcal{E}_i) \end{array}$$

avec $\mathbf{x}_i^0 = (x_i^0, y_i^0)$, et on notera par ailleurs $T^i_{\mathbf{x},\theta}$ la transformation plane $\tau_{\mathbf{x}} \circ R_{\theta-\theta_i^0} \circ \tau_{-\mathbf{x}_i^0}$ qui envoie l'ellipse de référence $\mathcal{E}_i^0$ sur $\mathcal{E}_i(\mathbf{x}, \theta)$. Ce qui nous permet de finalement définir le domaine de calcul Ω (figure 3.4), fonction des six paramètres x_1, y_1, θ_1, x_2, y_2 et θ_2 :

$$\begin{array}{rcl} \Omega : \quad E \subset \mathbb{R}^6 & \longrightarrow & \mathcal{O}\left(\mathbb{R}^2\right) \\ X = (x_1, y_1, \theta_1, x_2, y_2, \theta_2) & \longmapsto & \Omega(X) = \Omega_0 \backslash \left(\overline{\mathcal{E}_1(x_1, y_1, \theta_1)} \cup \overline{\mathcal{E}_0} \cup \overline{\mathcal{E}_2(x_2, y_2, \theta_2)}\right) \end{array} \tag{3.17}$$

Paramètres fixes			Variables
Ellipses de référence	Positions :	$x_1^0 = -7 \quad y_1^0 = -0.5$	$-10 \leq x_1 \leq -6.5$
		$x_2^0 = 7.5 \quad y_2^0 = -0.5$	$-1.5 \leq y_1 \leq 0$
	Paramètres angulaires :	$\theta_1^0 = -3° \quad \theta_2^0 = 3°$	$-10° \leq \theta_1 \leq 0°$
	Demis axes :	$A = 1 \quad a = 0.25$	$7.25 \leq x_2 \leq 10$
		$B = 5 \quad b = 0.5$	$-1.5 \leq y_2 \leq 0$
Domaine Ω_0	$L = 80 \quad l = 40$		$0° \leq \theta_2 \leq 10°$

Table 3.1: Valeurs des paramètres géométriques

L'ensemble E est un produit d'intervalles dont les bornes figurent dans la table 3.1.

On choisit ensuite un modèle $\mathcal{S}$ de fluide entre écoulement de type Stokes ou Navier-Stokes, dont nous préciserons les caractéristiques et les conditions aux bords dans la section dédiée 3.3.2. A tout domaine Ω, le modèle $\mathcal{S}(\Omega)$ associe un couple solution $(\mathbf{u}_\Omega, p_\Omega) \in H^1(\Omega)^2 \times L^2(\Omega)$. Ainsi, pour tout $X \in E$, la résolution de $\mathcal{S}(\Omega(X))$ aboutit à la solution $(\mathbf{u}_{\Omega(X)}, p_{\Omega(X)}) \in H^1(\Omega(X))^2 \times L^2(\Omega(X))$, dont on ne conservera plus que la dépendance vis-à-vis de X afin d'alléger les notations : on notera $(\mathbf{u}_X, p_X)$ la solution de $\mathcal{S}(X)$ dans $H^1(\Omega(X))^2 \times L^2(\Omega(X))$.

On se donne alors un jeu de paramètres cibles $X^* \in E$, pour lequel on notera p^* le champ de pression dans la solution de $\mathcal{S}(X^*) : p^* = p_{X^*}$. Ce champ de pression nous permet de définir formellement la fonctionnelle suivante, pour tout $X = (x_1, y_1, \theta_1, x_2, y_2, \theta_2) \in E$:

$$J(X) = \int_{\partial\mathcal{E}_0} |p_X - p^*|^2 + \int_{\partial\mathcal{E}_1^0} \left|p_X \circ T^1_{(x_1,y_1),\theta_i} - p^*\right|^2 + \int_{\partial\mathcal{E}_2^0} \left|p_X \circ T^2_{(x_2,y_2),\theta_i} - p^*\right|^2 \quad (3.18)$$

Ecrites telles quelles, ces intégrales présentent l'avantage d'être définies sur des bords statiques, mais on peut, de manière équivalente, les écrire sur les bords mobiles grâce aux changements de variable unitaires ${T^i_{(x_i,y_i),\theta_i}}^{-1}$:

$$\int_{\partial\mathcal{E}_i^0} \left|p_X \circ T^1_{(x_1,y_1),\theta_i} - p^*\right|^2 = \int_{\partial\mathcal{E}_i((x_i,y_i),\theta_i)} \left|p_X - p^* \circ {T^i_{(x_i,y_i),\theta_i}}^{-1}\right|^2$$

L'existence de cette intégrale est *a priori* incertaine puisque, que $\mathcal{S}$ soit le système d'équations de Stokes ou de Navier-Stokes, la pression est en général dans $L^2(\Omega)$ et n'admet donc pas nécessairement une trace de carré intégrable sur le bord. Cependant, d'après [13], compte tenu de la régularité de la frontière sur laquelle ces intégrales sont définies ainsi que des données du problème, le champ de pression possède au moins la régularité H^1, aussi bien pour le problème de Stokes que pour les équations de Navier-Stokes. Les intégrales apparaissant dans 3.18 sont donc parfaitement définies puisque $\forall X \in E,\ p_X|_{\partial\Omega(X)} \in L^2(\partial\Omega(X))$.

On définit ainsi une fonctionnelle qui présente un minimum global en X^* que l'on va essayer de retrouver numériquement en utilisant nos algorithmes de minimisation, en partant d'un point quelconque, bien évidemment différent de la cible X^*. La connaissance du point en lequel le minimum est réalisé permet de porter différentes considérations en rapport avec les performances de l'algorithme utilisé sur ce type de problème : calcul de distance entre les configurations d'ellipses obtenues et la configuration optimale, comparaison des champs de pression, courbes de convergence, *etc.*

3.3.2 Modèles de fluide

On modélise le fluide par les systèmes d'équations de Stokes ou de Navier-Stokes incompressibles. Le premier permet d'obtenir des résultats relativement vite puisqu'il s'agit

d'un système linéaire, et nous l'avons donc essentiellement utilisé pour la conception du script ainsi que pour obtenir nos premiers résultats. Le second, que l'on résout par une méthode de Newton, est beaucoup plus gourmand et demande un temps nettement plus long pour le calcul de J pour une unique valeur des paramètres. Pour les deux modèles de fluides, le problème est discrétisé avec des éléments de Taylor-Hood P^2 pour le champ de vitesse, et P^1 pour le champ de pression. On exploitera autant que possible les opportunités de réduction des coûts de calcul qu'apporte l'adaptation de maillage. La métrique utilisée pour ces systèmes est calculée par le procédé d'intersection exposé dans le chapitre 1 (section 1.4.2), afin d'obtenir un maillage épousant les variations de chacune des composantes des champs de pression et de vitesse. Nous verrons d'ailleurs que cela peut se révéler problématique et qu'il peut en résulter une fonctionnelle aux variations chaotiques si l'on ne prend pas quelques précautions.

Le système de Stokes

Il s'agit là d'une version simplifiée des équations de Navier-Stokes qui s'applique dans les cas où les effets visqueux prédominent sur les effets inertiels dans le fluide. C'est notamment le cas lorsqu'un fluide s'écoule dans un lieu étroit ou autour d'un objet de petite taille, ou encore pour des liquides très épais et visqueux tels que le miel. Sous leur forme forte usuelle, ces équations portent sur les champs de vitesse $\mathbf{u} \in C^2(\Omega)^2$ et de pression $p \in C^1(\Omega)$ du fluide qui sont reliés par les relations suivantes :

$$\left\{\begin{array}{rl} -\nu\Delta\mathbf{u} + \nabla p = \mathbf{0} & \\ \nabla.\mathbf{u} = 0 & \textit{dans } \Omega \\ \mathbf{u} = 0 & \textit{sur } \partial\mathcal{E}_i,\ i \in \{0,1,2\} \\ \mathbf{u} = \mathbf{g} & \textit{sur } \Gamma_1\ (1) \\ \big[-p\mathbb{I} + \nu\nabla\mathbf{u}\big]\mathbf{n} = \mathbf{0} & \textit{sur } \Gamma_2\ (2) \end{array}\right. \tag{3.19}$$

où $\mathbf{g} \in L^2(\Gamma_1)$ sera explicitée plus loin et $\{\Gamma_1, \Gamma_2\}$ forme une partition de la frontière du domaine $\partial\Omega_0$ (voir figure 3.4). Nous approfondirons la question des conditions aux limites dans la section qui leur est dédiée.

Ces équations sont considérées dans leur formulation faible afin de pouvoir les traiter par une méthode d'éléments finis. On définit les espaces fonctionnels suivants :

$$V = \Big\{ \mathbf{v} \in H^1(\Omega)^2 \ \Big|\ \mathbf{v} - \mathbf{g}|_{\Gamma_1} = \mathbf{0} \text{ et } \mathbf{v}|_{\partial\mathcal{E}_i} = \mathbf{0},\ \forall i \in \{1,2,3\} \Big\} \tag{3.20}$$

$$V_0 = \Big\{ \mathbf{v} \in H^1(\Omega)^2 \ \Big|\ \mathbf{v}|_{\Gamma_1} = \mathbf{0} \text{ et } \mathbf{v}|_{\partial\mathcal{E}_i} = \mathbf{0},\ \forall i \in \{1,2,3\} \Big\} \tag{3.21}$$

Dans sa formulation faible, le problème 3.19 consiste à trouver $(\mathbf{u}, p) \in V \times L^2(\Omega)$ tel que :

$$\forall (\mathbf{v}, q) \in V_0 \times L^2(\Omega)\ ,\ \int_\Omega \nu\,(\nabla\mathbf{u}, \nabla\mathbf{v}) - \int_\Omega p\nabla.\mathbf{v} - \int_\Omega q\nabla.\mathbf{u}\ =\ 0 \tag{3.22}$$

Une justification rigoureuse de cette formulation variationnelle ainsi que tout autre élément de théorie en rapport avec les équations de Stokes pourront être trouvés dans [45], notamment les arguments permettant de s'assurer que la solution de 3.22 vérifie 3.19 au sens des distributions, ainsi que les questions d'existence et d'unicité.

A noter que la solution de ce problème variationnel est la solution du problème de minimisation $\mathbf{u} = \underset{\mathbf{v}\in V}{\operatorname{argmin}} \int_\Omega \frac{\nu}{4} \left\| \nabla\mathbf{v} + \nabla\mathbf{v}^{\mathrm{T}} \right\|^2$, sous la contrainte $\mathbf{u} \in K = \{\mathbf{v} \in H^1(\Omega)\,|\,\nabla.\mathbf{v} = 0\}$. La formulation variationnelle 3.22 peut alors se voir comme une généralisation de la condition de Lagrange 2.4 dans laquelle p joue le rôle du multiplicateur de Lagrange.

Equations de Navier-Stokes incompressibles - Méthode de Newton

Le second modèle que nous utilisons est celui d'un fluide newtonien, incompressible et à l'équilibre, pourvu d'une viscosité cinématique ν. Avec des conditions aux limites identiques à celles apparaissant dans 3.19, la formulation classique associée à ce modèle correspond au système d'équations aux dérivées partielles non linéaires suivant :

$$\left\{\begin{array}{rl} -\nu\Delta\mathbf{u} - \mathbf{u}.\nabla\mathbf{u} + \nabla p = 0 & \\ \nabla.\mathbf{u} = 0 & \textit{dans } \Omega \\ \mathbf{u} = 0 & \textit{sur } \partial\mathcal{E}_i,\ i \in \{0,1,2\} \\ \mathbf{u} = \mathbf{g} & \textit{sur } \Gamma_1\ (1) \\ \left[-p\mathbb{I} + \nu\nabla\mathbf{u}\right].\mathbf{n} = \mathbf{0} & \textit{sur } \Gamma_2\ \textit{(2)} \end{array}\right. \tag{3.23}$$

En utilisant les espaces de fonctions 3.20 et 3.21, on peut établir une formulation variationnelle de ces équations (se référer à [45]) qui consiste à trouver un couple $(\mathbf{u}, p) \in V \times L^2(\Omega)$ tel que :

$$\forall(\mathbf{v}, q) \in V_0 \times L^2(\Omega)\ ,\ \int_\Omega \nu\left(\nabla\mathbf{u}, \nabla\mathbf{v}\right) + \int_\Omega \left(\mathbf{u}.\nabla\mathbf{u}\right).\mathbf{v} - \int_\Omega p\nabla.\mathbf{v} - \int_\Omega q\nabla.\mathbf{u} = 0 \tag{3.24}$$

Après discrétisation, cette formulation variationnelle n'aboutit pas sur un système linéaire à cause du terme non linéaire $\int_\Omega \left(\mathbf{u}.\nabla\mathbf{u}\right).\mathbf{v}$. La résolution de ce système va donc nécessiter une méthode spéciale pour traiter la non-linéarité. Nous utilisons la méthode de Newton qui s'applique ici comme suit :

Algorithme 3.1. Méthode de Newton appliquée à la formulation faible des équations de Navier-Stokes incompressibles :

- *On initialise* $(\mathbf{u}_0, p_0)$ *avec la solution du problème de Stokes 3.22 si* $\nu > 10^{-2}$, *sinon avec une solution des équations de Navier-Stokes associée à une viscosité cinématique* ν_0 *supérieure à* ν, *par exemple* $\nu_0 = 2\nu$.
- *Pour tout* $k \in \mathbb{N}$:

  — *trouver* $(\mathbf{w}_k, r_k) \in V_0 \times L^2(\Omega)$ *tel que* $\forall(\mathbf{v}, q) \in V_0 \times L^2(\Omega)$:
  $$\int_\Omega \left[\nu\left(\nabla\mathbf{w}_k, \nabla\mathbf{v}\right) + \left(\mathbf{w}_k.\nabla\mathbf{u}_k\right).\mathbf{v} + \left(\mathbf{u}_k.\nabla\mathbf{w}_k\right).\mathbf{v} - r_k\nabla.\mathbf{v} - q\nabla.\mathbf{w}_k\right]$$
  $$= \int_\Omega \left[\nu\left(\nabla\mathbf{u}_k, \nabla\mathbf{v}\right) + \left(\mathbf{u}_k.\nabla\mathbf{u}_k\right).\mathbf{v} - p_k\nabla.\mathbf{v} - q\nabla.\mathbf{u}_k\right]$$
  — Actualisation de l'itéré $(\mathbf{u}_{k+1}, p_{k+1}) = (\mathbf{u}_k - \mathbf{w}_k\ ,\ p_k - r_k)$

Dans cette méthode, on cherche à annuler l'application $\mathbf{\Phi}$ non linéaire qui à un élément $(\mathbf{u}, p)$ de $V \times L^2(\Omega)$ associe la forme linéaire $\mathbf{\Phi}(\mathbf{u}, p)$ sur $V_0 \times L^2(\Omega)$: $(\mathbf{v}, q) \mapsto \int_\Omega \nu\left(\nabla\mathbf{u}, \nabla\mathbf{v}\right) + \int_\Omega \left(\mathbf{u}.\nabla\mathbf{u}\right).\mathbf{v} - \int_\Omega p\nabla.\mathbf{v} - \int_\Omega q\nabla.\mathbf{u}$. L'équation variationnelle sur les corrections $(\mathbf{w}, r)$ correspond à l'égalité dans $\mathcal{L}\left(V_0 \times L^2(\Omega), \mathbb{R}\right)$ entre les formes linéaires $d\mathbf{\Phi}(\mathbf{u}, p)(\mathbf{w}, r)$ et $\mathbf{\Phi}(\mathbf{u}, p)$. Il s'agit de problèmes linéaires. Pour les calculs, il est bien sûr nécessaire de se donner un critère d'arrêt. On pourra par exemple choisir une précision $\epsilon > 0$ et une norme, puis interrompre les calculs dès que $\|\mathbf{w}_k\|^2 + \|p_k\|^2 < \epsilon$.

A propos des conditions aux bords...

Dans les deux modélisations, apparaissent deux types de conditions aux bords : une condition de Dirichlet, consistant à imposer la valeur du champ de vitesse sur la partie Γ_1 de $\partial\Omega_0$ et sur les frontières des ellipses $\partial\mathcal{E}_i$, ainsi qu'une condition un peu plus complexe sur une partie que l'on a appelée Γ_2, le bord sortant.

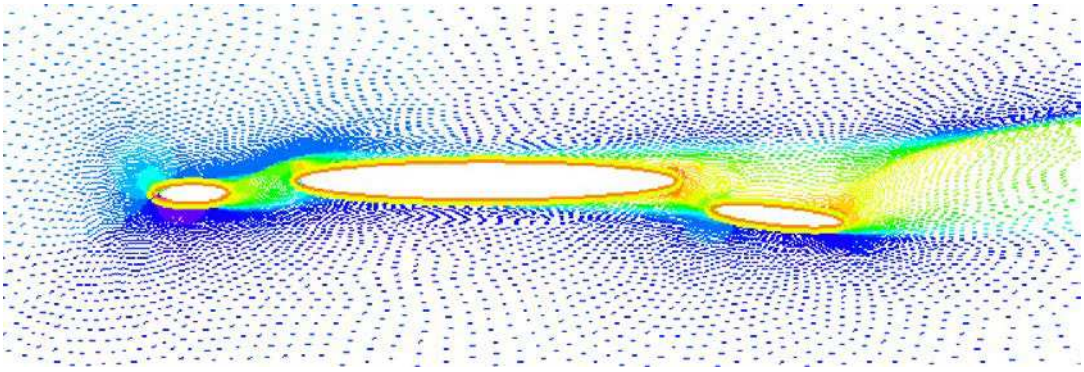


FIGURE 3.5: Champ de vitesse au voisinage des ellipses à $R_e = 100$

La condition de type Dirichlet est homogène sur les ellipses et traduit la présence de l'obstacle. Sur Γ_1, le reste de la frontière, la vitesse imposée est constante, de direction θ_∞ : $\mathbf{u}_\infty = u_\infty \begin{pmatrix} \cos\theta_\infty \\ \sin\theta_\infty \end{pmatrix}$. C'est cette condition non homogène qui imprime son mouvement au fluide.

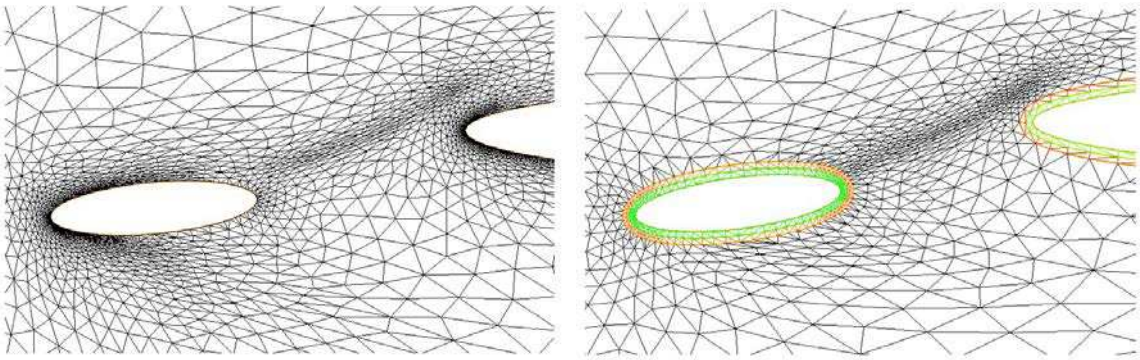
La seconde condition aux bords permet d'émuler un domaine d'extension infini par notre domaine fini. Celle-ci perturbe en effet très peu la forme de la solution à son voisinage. Dans le cas de notre problème, on observe que les ellipses engendrent un sillage, orienté selon l'angle d'attaque du fluide θ_∞, plus ou moins long selon la valeur de la viscosité. La condition $[-p\mathbb{I} + \nu\nabla\mathbf{u}]\,.\mathbf{n} = \mathbf{0}$ sur le bord droit permet à ce sillage de rencontrer le bord sans être modifié comme ce serait le cas en imposant une condition de Dirichlet non homogène.

3.3.3 Conséquences de l'adaptation de maillage

Nous avons observé, dans le cas des équations de Navier-Stokes résolues par la méthode de Newton avec adaptation de maillage, que la fonctionnelle 3.18 présentait de fortes oscillations, lorsque l'on laissait l'adaptation de maillage affecter l'intégralité du domaine de calcul, en particulier le voisinage des ellipses. Ces variations sont très chaotiques et perturbent les routines d'optimisation au point d'empêcher la récupération des paramètres ciblés avec une précision convenable.

Nous avons pu atténuer fortement ce bruit en ajoutant une couche d'éléments que l'on exclut du domaine affecté par la procédure d'adaptation de maillage. Ces mailles supplémentaires doivent être suffisamment fines pour ne pas compromettre la convergence des méthodes de Newton. On réalise donc en quelque sorte un compromis entre l'adaptation de maillage sur tout le domaine et une méthode qui consisterait à travailler sans adaptation, avec une triangulation assez fine pour que les calculs soient praticables. Une petite partie des économies en termes de degrés de liberté que l'on réalise avec l'adaptation de maillage est donc perdue, mais c'est le prix à payer pour ne pas trop perturber la fonctionnelle.

Nous n'avons pas d'analyse théorique solide quant à ce comportement de la fonctionnelle avec l'adaptation de maillage. Une piste de réflexion, plutôt spéculative, est que la frontière des ellipses, après avoir été maillée, n'est plus un objet géométrique lisse et devient un polygone non convexe. Or, d'après [13], le champ de pression dans ces conditions n'a plus la régularité suffisante pour que sa trace soit bien définie sur ces bords. Des singularités apparaissent, localisées sur les points anguleux de la frontière. Cette propriété n'est certes valable que pour les solutions de problèmes continus, mais il se peut que les solu-



(a) Maillage simple, intégralement adapté **(b)** Couches fixes autour des ellipses

Figure 3.6: Détails des maillages autour du voisinage de l'une des ellipses.

tions du problème discrétisé que nous calculons comportent des traces de ces singularités, et le fait d'être à la limite de la régularité requise est à l'origine des brusques variations entre les intégrales de pression calculées pour deux valeurs des paramètres même lorsque ces valeurs sont proches (voir figures 3.7).

3.3.4 Résultats d'optimisation

Les résultats numériques présentés ici sont ceux obtenus pour un fluide de Navier-Stokes avec une viscosité cinématique $\nu = 0.01$ (soit un nombre de Reynolds de 100). Les calculs sont effectués en parallèle sur la machine SGI du laboratoire Jacques-Louis Lions avec 40 processeurs et la version MPI de FreeFem++ pour l'optimisation avec CMA-ES, dont on a donc également utilisé l'interface MPI. Par défaut, l'implémentation de CMA-ES que nous utilisons manipule une population de $4 + \lfloor 3 \log n \rfloor$ individus, où n désigne la dimension de l'espace de recherche. Pour notre problème avec six paramètres, cette heuristique pour la taille de la population serait de neuf individus. Puisque l'on disposait de quarante processeurs, nous avons donc choisi d'augmenter la taille de la population à ce nombre, l'algorithme ne pouvant qu'être plus efficace avec une population plus étendue. Les autres paramètres de l'algorithme ne sont pas modifiés, en dehors d'un critère d'arrêt sur la tolérance de la fonction et l'évolution des variables d'optimisation.

Les calculs avec l'algorithme NEWUOA ont quant à eux été effectués séquentiellement sur un simple ordinateur portable Macintosh Macbook Pro de 2008. Celui-ci était équipé d'un processeur cadencé à 2.4 GHz et 3 Mo de mémoire cache. On utilise les "réglages" par défaut de l'algorithme, avec, là encore, une modification des critères d'arrêt pour les harmoniser avec ceux utilisés pour CMA-ES.

Les courbes de convergence 3.8 illustrent l'évolution de la valeur de la fonctionnelle au cours des deux processus d'optimisation. La méthode stochastique CMA-ES semble plus rapide au premier abord, mais il faut savoir que chacune des itérations portées en abscisse dissimule le calcul de quarante images par la fonction coût. La cinquantaine d'itérations nécessaire à la convergence de l'algorithme dans cette situation correspond en réalité à environ 2000 évaluations. Si ce calcul avait été pratiqué sur un processeur bi-coeur, comme pour le calcul avec NEWUOA, il aurait exigé une durée vingt fois plus longue.

Quoi qu'il en soit, si CMA-ES, méthode stochastique, demeure très gourmande en terme d'évaluations de la fonction coût, elle permet de récupérer les paramètres avec une précision bien meilleure que NEWUOA. Au reste, en superposant les configurations spatiales des ellipses obtenues avec celles correspondant aux paramètres ciblés, figure 3.9,

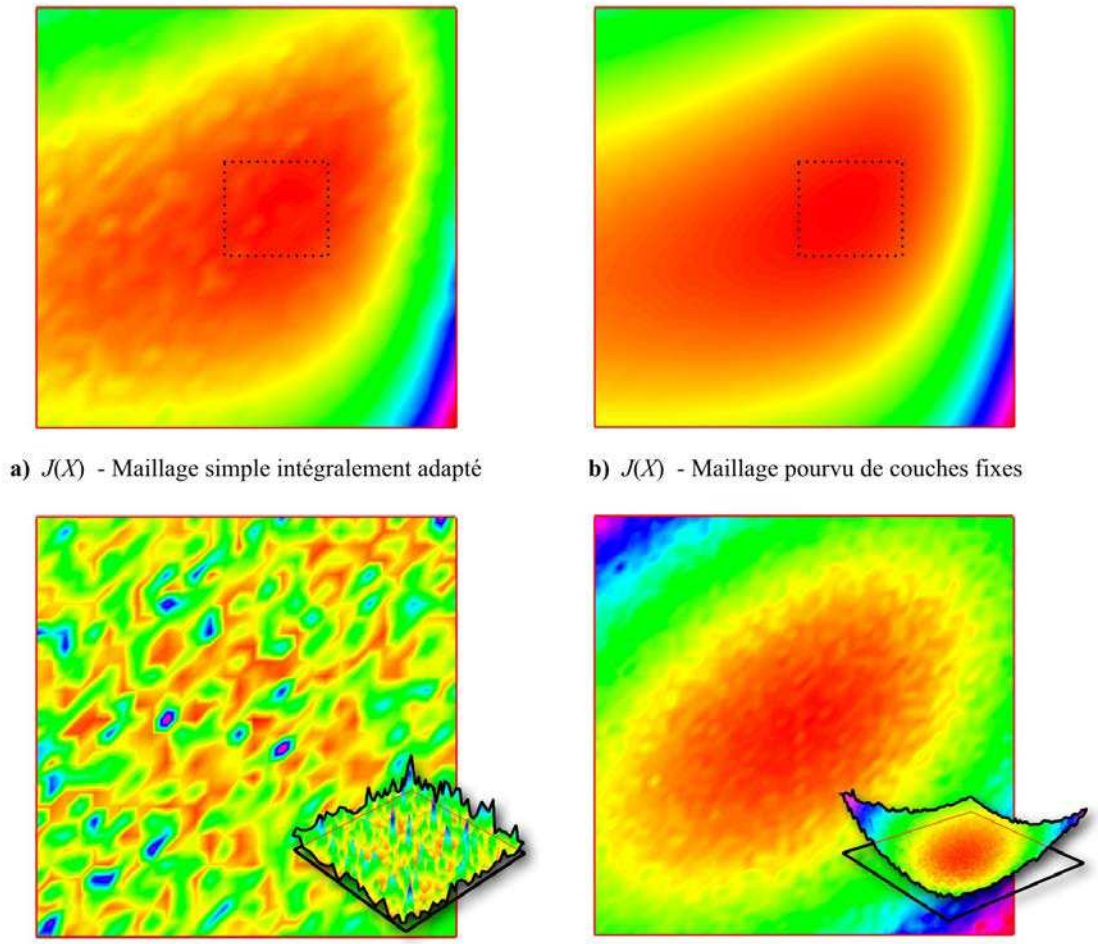


FIGURE 3.7: Représentation de la fonction coût J en fonction de y_1 et θ_1 et d'une valeurs fixe des autres paramètres pour les deux types de maillages.

on remarque que NEWUOA rencontre des difficultés à retrouver la position de l'ellipse la plus à droite, en particulier l'abscisse x_2 de son centre. Ce n'est pas surprenant car de petites translations de cette ellipse dans cette direction vont très peu affecter la trace du champ de pression, que ce soit sur son bord ou celui des autres. Le très léger angle implique en effet que cette ellipse se retrouve dans la traînée de l'ellipse centrale, quasiment invariante par rapport à x pour l'ordre de grandeur dans lequel x_2 peut varier (voir 3.5).

Nous reviendrons également sur ce problème dans le chapitre suivant dans lequel nous utiliserons notre version prototype de FreeFem++-ad, le noyau FreeFem++ enrichi de fonctionnalités de différences automatiques. Nous avons essayé de l'appliquer au cas du fluide de Navier-Stokes, mais les instabilités du code n'ont pas permis de mener ce calcul à bien. Cela a en revanche été possible avec le système de Stokes. Ce type de problème est l'exemple idéal pour l'application de la différentiation automatique. La fonction coût s'exprimant comme une intégrale sur un bord mobile, sa dérivée est relativement difficile à calculer. De plus, le petit nombre de paramètres de différentiation fait que les coûts additionnels en termes de temps de calcul et de mémoire resteront raisonnables pour une différentiation automatique en mode direct.

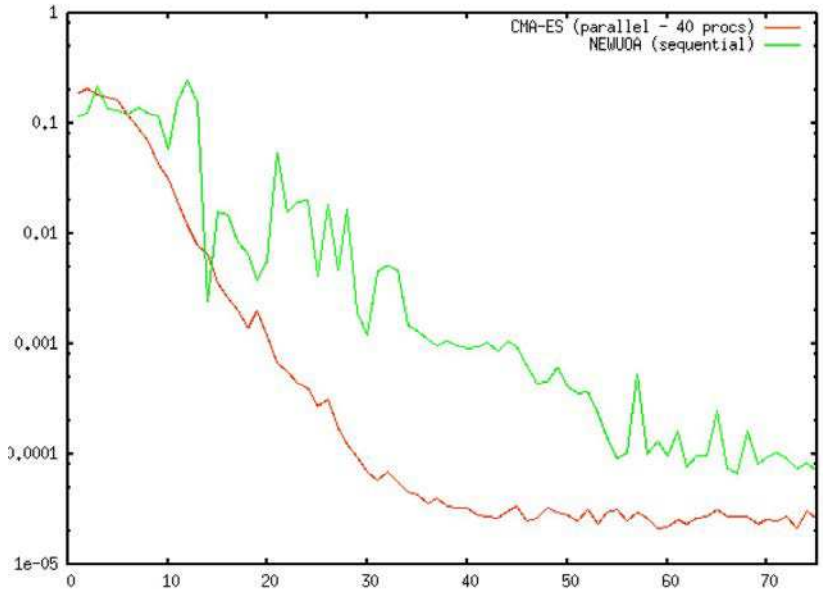


Figure 3.8: Evolution de la fonction coût au cours des minimisations avec CMA-ES et NEWUOA

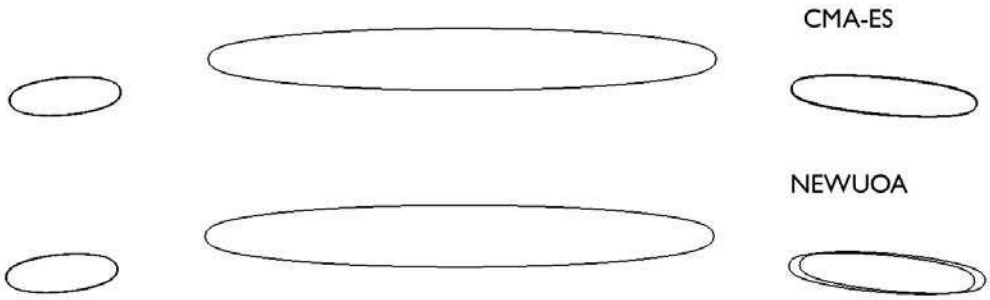


Figure 3.9: Superposition des configurations finales des ellipses avec la configuration cible

3.4 Simulation d'un condensat de Bose-Einstein

Au delà de sa valeur scientifique intrinsèque, la simulation d'un condensat de Bose-Einstein par minimisation de l'énergie de Gross-Pitaevskii constitue un problème particulièrement intéressant pour évaluer la robustesse d'une méthode d'optimisation car il s'agit d'un problème vraiment non linéaire, avec contrainte et impliquant le corps des nombres complexes. Sa résolution, quelle que soit la méthode utilisée, est difficile et il nous a paru intéressant de confronter IPOPT à ce problème, conjointement à une discrétisation par éléments finis avec adaptation de maillage. Cet essai constitue de plus un travail original de simulation numérique, l'utilisation de IPOPT dans ce cadre n'ayant pas fait, à ce jour l'objet de publications. L'étude [36], dans laquelle des minimiseurs sont obtenus par une autre méthode, mais toujours dans le cadre d'une discrétisation par éléments finis avec adaptation, fournit des éléments de comparaison et permet de confronter les résultats obtenus avec une nouvelle méthode d'optimisation.

Sans trop entrer en profondeur dans les méandres de la mécanique quantique, nous expliquerons succinctement ce que modélisent les objets mathématiques impliqués dans

l'énergie de Gross-Pitaevskii. Nous invitons les lecteurs particulièrement curieux et désireux de se forger une culture en mécanique quantique à consulter les ouvrages de Claude Cohen-Tannoudji [32] qui font référence en la matière. Des références bibliographiques plus spécifiques aux condensats de Bose-Einstein, que ce soit du point de vue du physicien expérimentateur que de celui du numéricien mathématicien pourront être trouvées dans [36], article pilier de notre étude.

3.4.1 Fonction d'onde et condensats de Bose-Einstein

Fonction d'onde pour une particule

En mécanique quantique non relativiste, l'évolution d'un système est régie par une *fonction d'onde* à valeurs dans un espace de Hilbert $\mathcal{H}$ appelé *espace des états* : $t \in \mathbb{R} \mapsto \Psi(t) \in \mathcal{H}$. L'espace $\mathcal{H}$ est construit à partir du système quantique étudié. On aura par exemple $\mathcal{H} = L^2(\mathbb{R}^3, \mathbb{C})$ pour une particule évoluant dans tout l'espace et dépourvue de tout autre caractéristique intrinsèque que sa masse. A $t \in \mathbb{R}$ fixé, la fonction $x \mapsto |\Psi(t,x)|^2$ de $\mathbb{R}^3$ dans $\mathbb{C}$ représente la densité de probabilité de présence de la particule. Autrement dit, la probabilité de trouver la particule dans un sous-domaine Ω de $\mathbb{R}^3$ à l'instant t est :

$$p(t,\Omega) = \int_\Omega |\Psi(t,x)|^2 \, dx$$

Et l'on doit avoir $\int_{R^3} |\Psi(t,x)|^2 \, dx = 1, \ \forall t$.

Par analogie avec les fluides, en écrivant une équation de conservation locale de la probabilité de présence de la particule, on peut définir un *courant de probabilité* $\mathbf{J}$ par $\nabla.\mathbf{J} = -\partial_t |\Psi|^2$. Cette grandeur s'explicite alors comme :

$$\mathbf{J} = \frac{i\hbar}{2m}\left(\Psi \nabla \overline{\Psi} - \overline{\Psi}\nabla\Psi\right) = \frac{\hbar}{2m}\left(i\Psi, \nabla\Psi\right) \tag{3.25}$$

où $(a,b) = a\bar{b} + \bar{a}b$ (défini sur n'importe quel produit d'ensembles $E \times F$ pour lesquels la conjugaison et la multiplication d'un élément de E par un élément de F ont un sens). Dans le cas d'une particule isolée, il est difficile de donner un sens concret à ce courant de probabilité puisque l'on ne saurait le rapprocher de la vitesse au sens classique qui joue déjà un rôle par l'intermédiaire de l'opérateur d'impulsion, et pour laquelle une transformée de Fourier de la fonction d'onde ci-dessus permet de calculer des densités de probabilité dans l'espace des impulsions. En revanche, il prend tout son sens dans le cas des superfluides et des condensats de Bose-Einstein pour lesquels il peut s'interpréter comme l'équivalent quantique du champ de vitesse locale du fluide.

Lorsque des observables à valeurs discrètes telles que le *spin* doivent être prises en compte, l'espace des états est le produit tensoriel du précédent espace fonctionnel par un espace, le plus souvent de dimension finie sur $\mathbb{C}$, décrivant la nouvelle observable. Pour une particule de spin 1/2 comme l'électron, le proton ou encore le neutron, on aura donc $\mathcal{H} = L^2(\mathbb{R}^3, \mathbb{C}) \otimes \mathbb{C}^2$. Si (e_-, e_+) est une base de C^2, le système est décrit à l'instant t par la fonction $x \in \mathbb{R}^3 \mapsto \Psi(t,x) = \Psi^-(t,x)e_- + \Psi^+(t,x)e_+$. La probabilité de trouver la particule dans un sous-domaine Ω avec une orientation du spin donnée $\pm$ est :

$$p(t,\Omega,\pm) = \int_\Omega |\Psi(t,x).e_\pm|^2 \, dx$$

avec la normalisation $\|\Psi^-\|_{L^2}^2 + \|\Psi^+\|_{L^2}^2 = 1$

L'évolution de la fonction d'onde est décrite par l'*équation de Schrödinger*, ici dans sa formulation la plus moderne :

$$i\hbar\frac{\mathrm{d}\Psi}{\mathrm{d}t} = H\Psi \tag{3.26}$$

où H est un opérateur hermitien linéaire de $\mathcal{H}$ dans lui-même appelé *hamiltonien*, associé à l'énergie totale du système.

Pour le cas de la particule libre sans spin, soumise à une force dérivant d'un potentiel $V : \mathbb{R}\times\mathbb{R}^3 \to \mathbb{R}$, l'hamiltonien s'écrit $H = -\dfrac{\hbar^2}{2m}\Delta + V$. On retrouve alors l'équation bien connue :

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\Delta\Psi + V\Psi \tag{3.27}$$

Pour un problème avec prise en compte des spins, l'équation 3.26 aboutit sur un système d'équations aux dérivées partielles. Un exemple est celui de l'équation de Pauli, cas particulier de l'équation de Schrödinger pour une particule de charge q et de spin $1/2$ évoluant dans un champ électromagnétique dérivant du potentiel $(\mathbf{A}, V)$:

$$\left[\frac{1}{2m}\left(\sigma.(-i\hbar\nabla - q\mathbf{A})\right)^2 + qV\right]\mathbf{\Psi} = i\hbar\frac{\partial\mathbf{\Psi}}{\partial t} \tag{3.28}$$

où $\sigma = \begin{pmatrix}\sigma_x\\ \sigma_y\\ \sigma_z\end{pmatrix}$ est un vecteur dont les composantes sont les matrices de Pauli (de dimension 2×2), et $\mathbf{\Psi} = \begin{pmatrix}\Psi^+\\ \Psi^-\end{pmatrix}$.

Fonction d'onde pour le problème à N corps

Les fonctions d'ondes et équations d'ondes associées précédemment exposées décrivent des systèmes quantiques très simples et permettent l'étude de particules isolées. Pour étudier des systèmes constitués par plusieurs particules, on doit travailler dans des espaces hilbertiens plus vastes. Pour N particules de même masse m, sans spin, on aura en effet $\mathcal{H} = \bigotimes_{i=1}^{N} L^2(\mathbb{R}^3, \mathbb{C})$, qui s'identifie à $L^2(\mathbb{R}^{3N}, \mathbb{C})$. L'équation de Schrödinger correspondante $i\hbar\dfrac{\partial\Psi}{\partial t} = -\dfrac{\hbar^2}{2m}\Delta\psi + V\Psi$, est une équation aux dérivées partielles en très grande dimension puisque $\Delta\Psi = \displaystyle\sum_{i=1}^{3N}\frac{\partial^2\Psi}{\partial x_i^2}$. Le potentiel V peut être très complexe selon les types de particules et les interactions modélisées puisque des termes d'appariement doivent parfois être pris en compte.

La prise en compte des spins des particules va largement compliquer la structure de $\mathcal{H}$ en ajoutant des espaces de dimension finie dans le produit tensoriel des espaces d'états. De plus, selon que le spin des particules est entier ou demi-entier, il est nécessaire de limiter l'espace des états à des fonctions d'ondes symétriques ou anti-symétriques, en vertu du *principe d'exclusion de Pauli*.

Condensat de Bose-Einstein - Energie de Gross-Pitaevskii

Un condensat de Bose-Einstein est un gaz d'atome se comportant comme des bosons (spin entier) dont la température est telle que la plupart de ses particules occupent le même état d'énergie. Dans cet état, on observe à l'échelle macroscopique des phénomènes

d'origine quantique très surprenants tels que la superfluidité, d'étranges réactions face aux stimulations extérieures, *etc.* Notre étude consiste à simuler un de ces condensats confiné par un potentiel quadratique, auquel on impose un mouvement de rotation. Des expériences en laboratoire montrent qu'un réseau très ordonné de vortex apparaît ([69]), phénomène que l'on espère obtenir numériquement avec IPOPT, comme il est fait avec d'autres techniques dans [36].

Strictement parlant, il faudrait étudier un tel gaz avec l'équation d'onde symétrique du paragraphe précédent, avec un potentiel reflétant la prise en compte des interactions entre les atomes. Cependant, si petit soit-il, un condensat de Bose-Einstein reste un système composé de milliers d'atomes. Il est donc inutile d'espérer pouvoir le modéliser numériquement par l'équation précédente. C'est là qu'interviennent les physiciens qui, avec l'aide de la physique statistique et de diverses approximations montrent que le système, à l'équilibre, peut être modélisé par une fonction, toujours à valeurs complexes mais des trois seules variables d'espace (ou deux pour une simulation bidimensionnelle), dont le carré du module correspond à la densité d'atome. Cette fonction d'onde minimise l'*énergie de Gross-Pitaevskii* qui s'écrit, après adimensionnement :

$$E(u) = \int_{\mathbb{R}^d} \frac{1}{2} |\nabla u|^2 + V|u|^2 + \frac{g}{2}|u|^4 - \frac{1}{2}\mathbf{\Omega} \wedge \mathbf{x}.\,(iu, \nabla u) \tag{3.29}$$

Dans cette expression, $\mathbf{\Omega} = \Omega \mathbf{e}_z$ est le vecteur rotation adimensionné, $\mathbf{x}$ est le point courant et g est une constante de couplage associée aux interactions entre les atomes. Enfin, V est le potentiel de confinement qui peut prendre la forme quadratique $V = \frac{1}{2}\left(\omega_x^2 x^2 + \omega_y^2 y^2 + \omega_z^2 z^2\right)$ lorsque le confinement est assuré par un champ magnétique, les ω_α désignant les fréquences spatiales, ou encore $V = (1-\alpha)r^2 + \frac{1}{4}kr^4 + \frac{1}{2}\omega_z^2 z^2$ quand le piégeage du condensat est amplifié par ajout d'un faisceau laser le long de l'axe z, ce qui permet d'imprimer à celui-ci des vitesses de rotation plus importantes (le cas du potentiel quadratique précédent limitant Ω aux valeurs strictement inférieures à $\min(\omega_x, \omega_y)$). Le terme d'ordre quatre provient dans ce dernier cas du développement en série d'un potentiel de la forme $U\exp(-2r^2/\omega^2)$, où r désigne la coordonnée radiale dans un système de repérage cylindrique.

La fonction u, pour satisfaire au principe de conservation de la masse, doit vérifier $\int_{\mathbb{R}^d} |u|^2 = 1$. Les détails de la dérivation de cette énergie pourront être trouvés dans [68]. L'analyse mathématique [85] suggère également qu'en considérant que l'extension spatiale du condensat dans la direction de l'axe de rotation est grande devant son épaisseur dans les autres directions, on peut se ramener à un problème bidimensionnel.

3.4.2 Minimisation de l'énergie de Gross-Pitaevskii avec FreeFem++ et IPOPT

Position du problème

Etant données les constantes réelles B, g et Ω, il s'agit de résoudre le problème de la minimisation de 3.29 en dimension $d = 2$ ou 3, sous la contrainte $\int_{\mathbb{R}^d} |u|^2 = 1$. En raison de la forte décroissance de $|u|$ avec la distance de l'axe de rotation du condensat, on peut restreindre le calcul à un domaine ouvert fini $\mathcal{D}$ de $\mathbb{R}^2$ sans que la solution déterminée diffère trop de celle du problème initial. Nous allons donc chercher à approcher numériquement les solutions du problème suivant :

Problème 3.2. *Trouver $u \in H_0^1(\mathcal{D}, \mathbb{C})$ telle que $\int_{\mathcal{D}} |u|^2 = 1$ et minimisant la fonctionnelle :*

$$E(u) = \int_{\mathcal{D}} \frac{1}{2} |\nabla u|^2 + V|u|^2 + \frac{g}{2}|u|^4 - \frac{1}{2}\mathbf{\Omega} \wedge \mathbf{x}.\,(iu, \nabla u)$$

On associe le minimum global de cette fonctionnelle à la configuration quantique fondamentale du condensat. Des solutions dont l'optimalité ne se vérifie que localement correspondent à des états excités métastables, le plus souvent de faible énergie compte tenu des hypothèses formulées.

On peut également écrire formellement les différentielles de cette énergie, qui coïncideront avec les expressions du gradient et de la hessienne de la version discrétisée. Pour toutes fonctions test v et w appartenant à $H_0^1(\mathcal{D},\mathbb{C})$:

$$\begin{aligned} dE(u)(v) &= \int_{\mathcal{D}} \left(\frac{1}{2}\left(\nabla u, \nabla v\right) + V(u,v) + g|u|^2(u,v) \right. \\ &\qquad \left. -\frac{1}{2}\mathbf{\Omega} \wedge \mathbf{x}.\left[(iv,\nabla u) + (iu,\nabla v)\right] \right) \quad (3.30) \\ d^2E(u)(v,w) &= \int_{\mathcal{D}} \left(\frac{1}{2}\left(\nabla w, \nabla v\right) + V(w,v) + g|u|^2(w,v) + g(u,w)(u,v) \right. \\ &\qquad \left. -\frac{1}{2}\mathbf{\Omega} \wedge \mathbf{x}.\left[(iv,\nabla w) + (iw,\nabla v)\right] \right) \quad (3.31) \end{aligned}$$

Discrétisation

Soit $(\mathcal{T}_h)_{h>0}$ une famille régulière de triangulations de $\mathcal{D}$ et $\mathcal{D}_h = \bigcup_{T\in\mathcal{T}_h} T$. Nous introduisons l'espace d'approximation de $H^1(\mathcal{D},\mathbb{R})$ par éléments finis :

$$W_h^l = \left\{ w \in C^0\left(\overline{\mathcal{D}_h},\mathbb{R}\right) \ ; \ \forall T \in \mathcal{T}_h, \ w|_T \in P^l\left(T,\mathbb{R}\right) \right\} \quad (3.32)$$

ainsi que :

$$V_h^l = \left\{ v \in W_h^l \ ; \ v|_{\partial\mathcal{D}_h} = 0 \right\} \quad (3.33)$$

V_h^l est un sous-espace de $H_0^1(\mathcal{D})$ de dimension finie à partir duquel nous dérivons une version discrétisée de l'énergie 3.2 $E_h^l : V_h^l \times V_h^l \to \mathbb{R}$:

$$E_h^l(u_r,u_i) = \int_{\mathcal{D}_h} \left(\frac{1}{2}|\nabla u_r|^2 + \frac{1}{2}|\nabla u_i|^2 + V(u_r^2+u_i^2) + \frac{g}{2}\left|u_r^2+u_i^2\right|^2 - \mathbf{\Omega}\wedge\mathbf{x}.\left(u_r\nabla u_i - u_i\nabla u_r\right) \right) \quad (3.34)$$

Fonctionnelle que l'on minimisera sous la contrainte :

$$C_h^l(u_r,u_i) = \int_{\mathcal{D}_h} u_r^2 + u_i^2 \quad (3.35)$$

L'utilisation d'espaces de fonctions à valeurs réelles est motivée par le fait que les algorithmes de minimisation ne travaillent en général qu'avec des paramètres réels. Si on note $(\varphi_r^k,\varphi_i^k)$, pour $1 \le k \le \dim(V_h^l \times V_h^l)$, les fonctions de base de notre espace d'éléments finis, et en introduisant les parties réelles et imaginaires dans les expressions 3.30, nous obtenons les formules suivantes pour les composantes du gradient et de la hessienne de 3.34 :

$$\begin{aligned} &\partial_j E_h^l(u_r,u_i) = \\ &\int_{\mathcal{D}_h} \left(\nabla u_r.\nabla\varphi_r^j + \nabla u_i.\nabla\varphi_i^j + 2V(u_r\varphi_r^j + u_i\varphi_i^j) + g(u_r^2+u_i^2)(u_r\varphi_r^j + u_i\varphi_i^j) \right. \quad (3.36) \\ &\qquad \left. -\mathbf{\Omega}\wedge\mathbf{x}.\left(\varphi_r^j\nabla u_i + u_r\nabla\varphi_i^j - \varphi_i^j\nabla u_r - u_i\nabla\varphi_r^j\right) \right) \end{aligned}$$

$$\begin{aligned}\partial_{jk}E_h^l(u_r,u_i) = \int_{\mathcal{D}_h} \nabla\varphi_r^j.\nabla\varphi_r^k + \nabla\varphi_i^j.\nabla\varphi_i^k + 2V(\varphi_r^j\varphi_r^k + \varphi_i^j\varphi_i^k) \\ + \int_{\mathcal{D}_h} g(u_r^2+u_i^2)(\varphi_r^j\varphi_r^k + \varphi_i^j\varphi_i^k) \\ + \int_{\mathcal{D}_h} 2g(\varphi_r^j u_r + \varphi_i^j u_i)(\varphi_r^k u_r + \varphi_i^k u_i) \\ - \int_{\mathcal{D}_h} \mathbf{\Omega}\wedge\mathbf{x}.\left(\varphi_r^k\nabla\varphi_i^j + \varphi_r^j\nabla\varphi_i^k - \varphi_i^k\nabla\varphi_r^j - \varphi_i^j\nabla\varphi_r^k\right)\end{aligned} \tag{3.37}$$

La contrainte 3.35 étant une forme quadratique homogène, on peut en calculer les gradients à partir de la simple multiplication de sa matrice hessienne par le vecteur des composantes de (u_r, u_i) dans la base de fonctions $(\varphi_r^j, \varphi_i^j)$:

$$\partial_{jk}C_h^l = \int_{\mathcal{D}_h} 2(\varphi_r^j\varphi_r^k + \varphi_i^j\varphi_i^k) \tag{3.38}$$

Encore une fois, FreeFem++ permet une transposition immédiate de ces expressions sous forme variationnelle à l'aide des commandes d'intégration `int2d` et de la déclaration de formulations variationnelles `varf`.

Approximation numérique d'un minimiseur avec IPOPT

Notre méthode consiste à minimiser directement l'énergie 3.2 après discrétisation du problème. On utilisera pour cela le logiciel IPOPT qui implémente la méthode de points intérieurs décrite dans la section 2.5. L'absence de contrainte d'inégalité fait que l'algorithme ne se comporte pas réellement comme une méthode de barrière, mais simplement comme une méthode de Newton avec recherche linéaire. Cette dernière favorise la convergence quelle que soit la fonction d'onde choisie pour l'initialisation de l'algorithme et permet de se dispenser de la continuation sur Ω pour les simulations associées à des valeurs élevées de la vitesse de rotation. IPOPT se révèle donc très efficace dans la résolution de ce problème

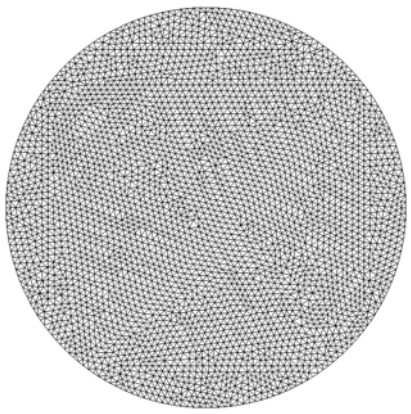
L'adaptation de maillage est encore une fois très opportune pour ce problème. L'étude [36] précise que le choix optimal de fonctions pour le calcul des métriques est de construire une intersection à partir des deux fonctions u_r et u_i. L'adaptation du maillage est d'autant plus utile que la solution est souvent nulle sur une part assez conséquente du domaine et présente une alternance de zones où elle varie peu et d'autres dans lesquelles les variations sont abruptes (notamment dans les vortex, ou à la surface du condensat). Il est donc très important de bien répartir les degrés de liberté.

On choisit une tolérance finale ϵ_f que l'on souhaiterait obtenir sur la solution calculée par IPOPT (voir section 2.5 pour des précisions sur le critère d'arrêt en tolérance). Comme suggéré dans [36], on initialise les parties réelles et imaginaires par la densité de Thomas-Fermi, qui ne dépend que de $r = |\mathbf{x}|$:

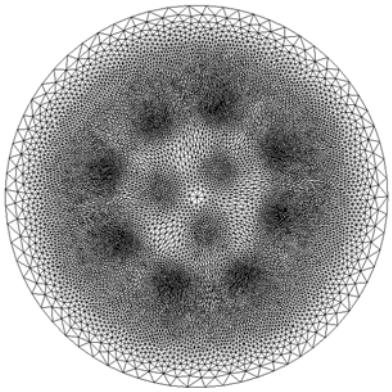
$$\rho_{\mathrm{TF}} = |u_{\mathrm{TF}}|^2 = \left(\frac{\mu - V}{g}\right)^+ \tag{3.39}$$

que l'on obtient par minimisation de l'énergie de Thomas-Fermi :

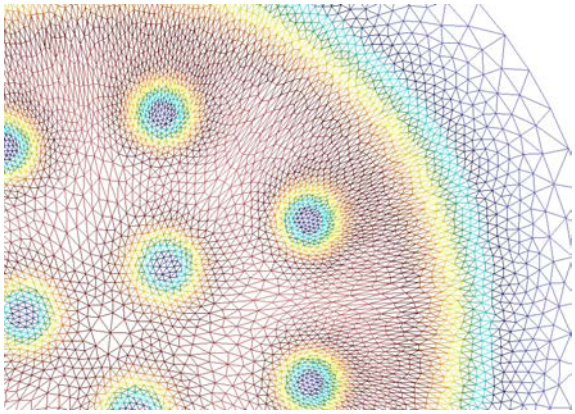
$$E_{\mathrm{TF}}(u) = \int_{\mathcal{D}} V|u|^2 + \frac{g}{2}|u|^4$$



(a) Maillage initial



(b) Maillage final



(c) Voisinage d'un vortex

Figure 3.10: Maillage 2D initial et maillage final pour $g = 500$ et $\Omega = 2$

et dans laquelle le scalaire μ joue le rôle de multiplicateur de Lagrange pour la contrainte $\int_{\mathcal{D}} |u|^2 = 1$. On pourra également initialiser l'optimisation par le résultat d'une minimisation précédente.

L'expression 3.39 permet également de déterminer le domaine de calcul $\mathcal{D}$: en remplaçant V par le potentiel effectif $V(\mathbf{x}) - \frac{\Omega^2 \mathbf{x}^2}{2}$, on obtient une densité $\rho^{\Omega}_{\mathrm{TF}}$ qui ne dépend elle aussi que de r, et qui permet de choisir pour $\mathcal{D}$ le disque ouvert de rayon $\kappa r^{\Omega}_{\mathrm{TF}}$, où $\kappa > 0$ et $r^{\Omega}_{\mathrm{TF}} = \sup \left\{ r \in \mathbb{R}^+, \rho^{\Omega}_{\mathrm{TF}}(r) > 0 \right\}$.

On choisit ensuite un niveau d'adaptation de maillage $N_a \in \mathbb{N}$, ainsi que le nombre d'itérations N_{iter} que l'on permettra d'effectuer à IPOPT tant que l'optimisation n'est pas pratiquée avec le maillage le plus fin. Ceci permet d'éviter de trop nombreuses itérations sur des maillages partiellement adaptés à la solution. Ce critère d'arrêt sur le nombre d'itérations est relâché lorsque l'on a adapté N_a fois le maillage afin qu'IPOPT ne s'arrête que lorsque la précision ϵ_f ou bien le nombre maximum d'itérations par défaut N^0_{iter} ont été atteints. Les grandes lignes de notre méthode sont explicitées dans l'algorithme 3.

La boucle de continuation sur la vitesse de rotation du condensat pourra éventuellement être remplacée par une boucle sur le terme d'interaction atomique g. Quoi qu'il en soit, si ces boucles sont incontournables lors de l'utilisation d'algorithmes à convergence locale, elles représentent une option lors de l'utilisation d'IPOPT, uniquement intéressante pour obtenir des résultats associés à de multiples valeurs de ces paramètres.

Le script FreeFem++ que nous proposons est inclus en annexe B. A noter que les tests d'arrêts relatifs à la boucle d'adaptation diffèrent légèrement de ceux figurant dans

Algorithm 3 Approximation numérique d'un minimiseur de 3.29 en couplant IPOPT et adaptation de maillage

$n_a \leftarrow 0$
$r \leftarrow -1$ *(le résultat renvoyé par IPOPT)*
$\lambda \leftarrow 1$ *(multiplicateur de Lagrange)*
$\mathcal{T}_h$ *est initialisé par triangulation de Delaunay du cercle de rayon* $\kappa r_{\mathrm{TF}}^{\Omega}$
$u_r + iu_i \leftarrow \sqrt{\rho_{\mathrm{TF}}}$
for $\Omega = \Omega_i$ **to** $\Omega = \Omega_f$ **do**
  while $n_a < N_a$ *et* $r \neq 0$ **do**
    $\mathcal{T}_h \leftarrow \mathrm{adaptmesh}\,(\mathcal{T}_h, (u_r, u_i))$
    u_r *et* u_i *sont remplacées par leurs interpolées sur le nouveau maillage*
    if $n_a \geq N_a - 1$ **then**
      $n_{\mathrm{iter}} \leftarrow N_{\mathrm{iter}}^0$
    else
      $n_{\mathrm{iter}} \leftarrow (1 + n_a) N_{\mathrm{iter}}$
    end if
    $r \leftarrow \mathrm{IPOPT}\,(E_h, C_h, (u_r, u_i), \lambda, \mathrm{maxiter} = n_{\mathrm{iter}}, \mathrm{tol} = \epsilon_f)$
      u_r, u_i *et* λ *sont également actualisés pendant cette étape*
  end while
  $\Omega \leftarrow \Omega + \delta\Omega$
end for

l'algorithme 3. Ce script pourrait faire l'objet de quelques optimisations, notamment par le calcul des matrices associées aux termes quadratiques de l'énergie et de la contrainte une seule fois pour chaque adaptation du maillage. L'adaptation du rayon du domaine de calcul à celui de l'énergie de Thomas-Fermi pourrait aussi être effectuée à chaque changement de Ω, ce qui éviterait d'avoir un domaine trop étendu pour les calculs associés aux valeurs les plus petites de ce paramètre. Ces modifications n'apportent pas de difficulté particulière mais compliqueraient cependant un script dont la lecture n'est déjà pas des plus agréables. Aussi n'avons-nous pas jugé vital de les y apporter, bien qu'elles soient effectivement codées dans les programmes que nous avons utilisés pour les résultats suivants.

Résultats

Nous nous basons sur [36] pour établir une série de tests pour lesquels nous disposerons d'éléments de comparaison. Tous les calculs, que ce soit en dimension deux ou trois, sont effectués sur un ordinateur portable récent équipé d'un processeur Intel Core i7 cadencé à 2.8 GHz et disposant d'une mémoire cache de 8Mo, avec 16 Go de mémoire vive. Il était également possible d'utiliser pour cette tâche la machine HPC du laboratoire, cependant, le code s'exécutant purement séquentiellement, cette alternative n'était pas réellement avantageuse puisque les processeurs de cette machine, plus anciens, sont beaucoup moins rapides (2 Ghz). En dépit d'une mémoire cache plus importante, les calculs menés sur ce dernier se sont révélés plus laborieux. C'est pourquoi ils ont finalement été effectués sur un ordinateur personnel. IPOPT se montre d'une efficacité certaine pour ce problème en révélant des performances très avantageuses au regard de la méthode employée dans [36] pour la minimisation de l'énergie de Gross-Pitaevsky en approximation par éléments finis.

Pour ces calculs en dimension deux, en dehors des derniers tests à forte constante d'interaction atomique, le potentiel de confinement du condensat est la somme d'un potentiel

harmonique et d'un potentiel quartique :

$$V = \frac{1}{2}(x^2 + y^2) + \frac{1}{4}(x^2 + y^2)^2$$

Il a déjà été question de ce potentiel dont la particularité est de lever la limite imposée par le potentiel harmonique sur la vitesse de rotation du condensat. On peut donc dans ce cas, en théorie, mener des calculs pour toutes valeurs de Ω.

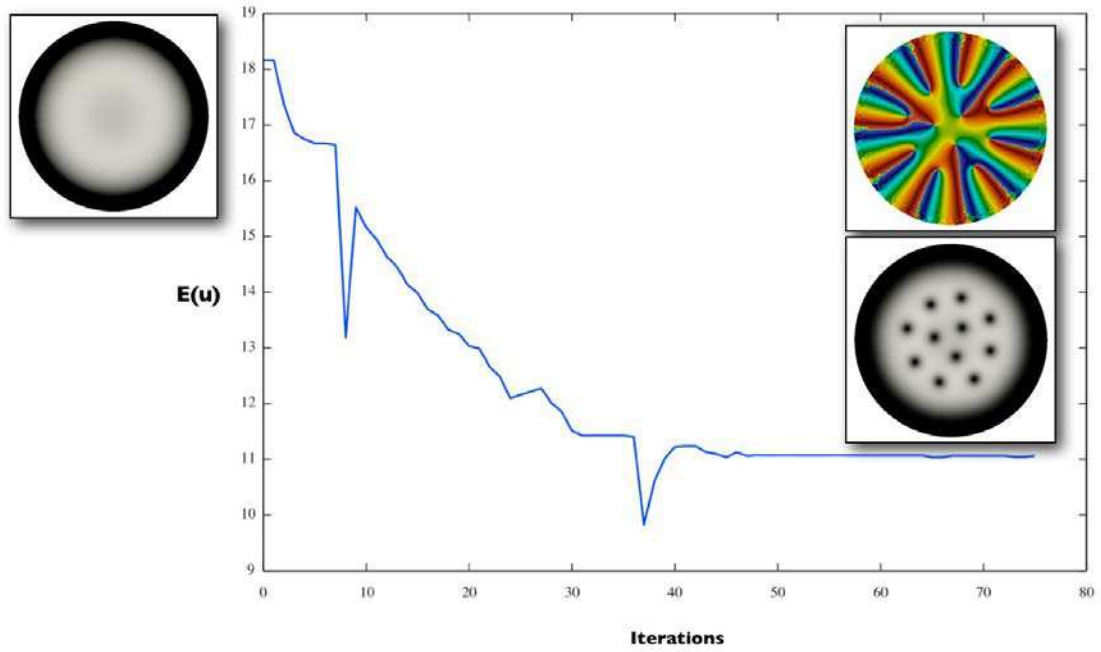


Figure 3.11: Evolution de l'énergie au cours du processus d'optimisation pour $g = 500$ et $\Omega = 2$, en dimension 2

On commence par effectuer un calcul sur un cas simple pour lequel la constante d'interaction atomique g et la vitesse de rotation ω sont faibles, avec les valeurs $g = 500$ et $\Omega = 2$. En dimension deux, il s'agit d'un calcul relativement rapide dont l'exécution ne requiert que quelques minutes pour peu que la méthode employée soit raisonnablement efficace. IPOPT converge après environ soixante-dix itérations et deux adaptations de maillage successives, en un temps certes beaucoup plus long que les méthodes de gradient de Sobolev ou de propagation en temps imaginaire (mentionnées dans [36]), mais avec un maillage comportant environ dix fois plus de points.

	M	**N_t/N_v**	**E(u)**	**Itérations**	**CPU (s)**
Gradient de Sobolev [36]	200	3722/ ?	11.87	232	55
Temps Imaginaire [36]			11.87	139	54
IPOPT	200	24462/12332	11.66	75	179

Table 3.2: $g = 500$ et $\Omega = 2$ en 2D

Les maillages utilisés en début et fin d'optimisation sont représentés sur les figures 3.10. La courbe 3.11 illustre la convergence de l'algorithme. On peut également y voir la densité finale, ainsi que les variations de la phase de la fonction d'onde déterminée. On observe bien le changement de phase centré autour des vortex, révélateur de zone de

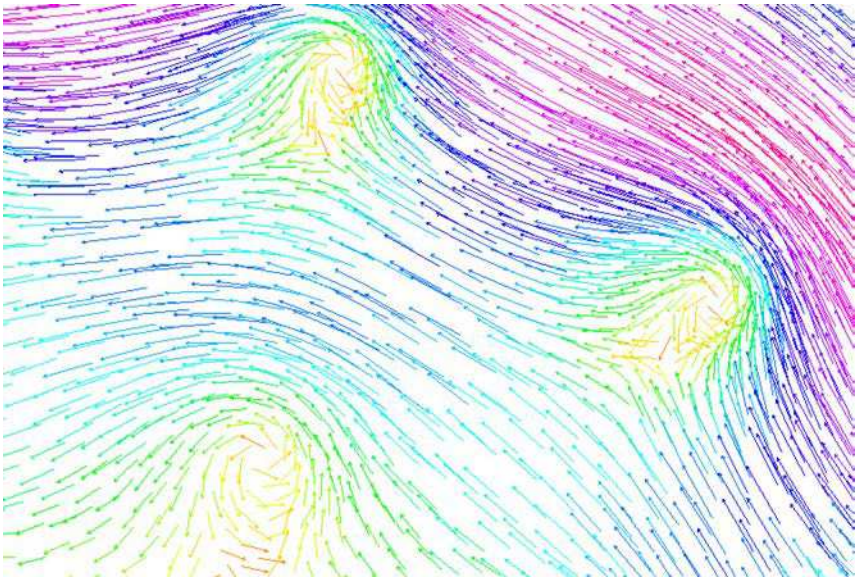


FIGURE 3.12: Courant de probabilité au voisinage de quelques vortex pour $g = 500$ et $\Omega = 2$

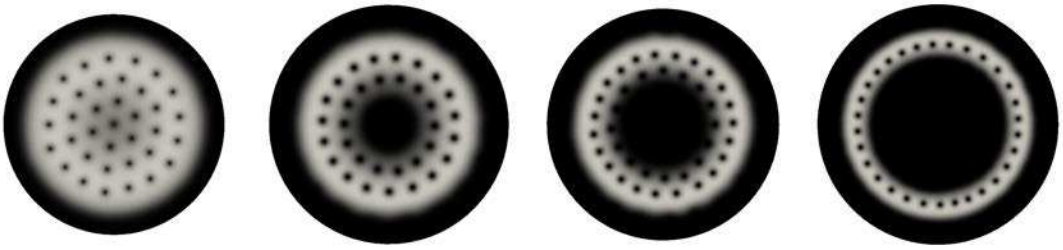
vorticité. On observe plus ou moins cet enroulement des lignes de champ du courant de probabilité défini en 3.25 sur la figure 3.12.

Suivant les expérimentations de [36], on augmente légèrement la vitesse de rotation jusqu'à $\Omega = 2.5$, tout en conservant tous les autres paramètres à la même valeur. Dans l'étude [36], le calcul pour $\Omega = 2.5$ est initialisé par la fonction d'onde obtenue pour $\Omega = 2$. Au regard des différences topologiques que présentent les solutions obtenues pour ces deux valeurs de Ω, nous pensons qu'une telle initialisation ne permet pas de réduire sensiblement la durée du calcul par rapport à une initialisation avec le minimiseur de l'énergie de Thomas-Fermi, surtout lorsque l'on utilise un algorithme à convergence globale tel qu'IPOPT. Il est même certain que comme les premières itérations pendant lesquelles se profile la solution sont effectuées sur un maillage comportant plus de points dans le premier cas, sans être pour autant adapté à la solution finale, cet initilisation engendre une phase de calculs superflue. Les calculs menés avec les deux initialisations corroborent cette intuition (voir table 3.3). Il apparaît clairement que bien qu'initialiser avec le résultat de l'optimisation précédente permette d'économiser quelques itérations, la surcharge de calcul introduite par le maillage initial mal adapté allonge considérablement le temps de calcul. [36] affiche une durée de calcul inférieure, mais pour un maillage final de moindre finesse, puisque celui-ci comporte moitié moins de points. La valeur de l'énergie n'atteint pas une précision aux mêmes décimales.

	Initialisation	**M**	$\mathbf{N_t/N_v}$	**E(u)**	**Itérations**	**CPU (s)**
IPOPT	Thomas-Fermi	200	14942/29782	5.76	98	487
	$\Omega = 2$ et $g = 500$	200	17560/34954	5.75	88	590
Sobolev [36]	$\Omega = 2$ et $g = 500$	200	8968/-	6.08	1266	321

TABLE 3.3: Résumé de l'optimisation pour $g = 500$ et $\Omega = 2.5$ et différentes initialisations en 2D

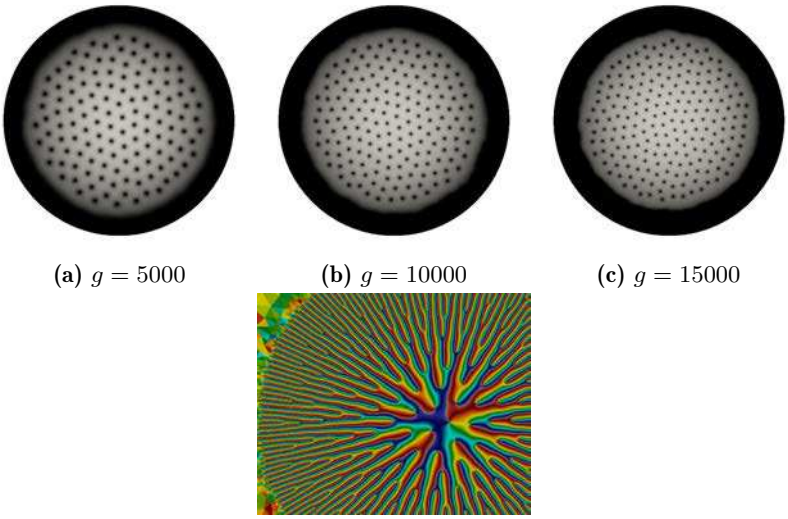
Nous avons également mené une série de calculs pour observer le comportement de la solution associée à de hautes valeurs de la vitesse de rotation et établir si l'algorithme convergeait bien vers les états de vortex géants obtenus dans les travaux qui ont précédés les nôtres. Toujours pour suivre [36], la constante g est relevée à la valeur 1000 afin d'aug-



(a) $\Omega = 3$ **(b)** $\Omega = 3.5$ **(c)** $\Omega = 4$ **(d)** $\Omega = 5$

Figure 3.13: Densités associées aux minimiseurs calculés pour $g = 1000$ et $\Omega = 3, 3.5, 4$ et 5

menter l'effet de la partie vraiment non-linéaire de l'énergie, et ainsi rendre le calcul plus délicat, ce qui permet d'éprouver plus rigoureusement notre méthode. Grâce à l'adaptation de maillage, on peut voir sur les figures 3.13 que les zones de densité nulle sont bien capturées tout autour du vortex géant central.



(a) $g = 5000$ **(b)** $g = 10000$ **(c)** $g = 15000$

(d) Variations de la phase pour $g = 15000$

Figure 3.14: Densités associées au minimiseurs calculés pour le potentiel harmonique pur, $\Omega = 0.95$ et $g = 5000,\ 10000$ et 15000

Enfin, l'ultime test pour le cas de dimension deux est celui de hautes valeurs de la constante d'interaction atomique, avec un potentiel de piégeage purement harmonique $V = \frac{1}{2}(x^2 + y^2)$ et une rotation proche de la limite au-delà de laquelle le condensat perd sa cohésion $\Omega = 0.95$. Dans ce cas, il a été observé expérimentalement que le condensat adopte une structure avec un très grand nombre de vortex, évoquant la forme d'un nid de guêpes, avec des trous d'autant plus nombreux et fins que g est élevé (voir figure 3.14). Numériquement, l'adaptation du maillage à une telle solution va demander un grand nombre de points pour pouvoir suivre les variations de la fonction d'onde avec la précision nécessaire à la convergence de notre algorithme. Il s'agit donc de calculs relativement lourds (pour de la dimension 2) qui peuvent prendre plusieurs heures. L'efficacité de l'algorithme utilisé est alors déterminante.

	Ω	M	N_t/N_v	E(u)	Itérations
IPOPT	3	200	33558/67038	2.72	151
	3.5	200	45605/91141	−12.03	164
	4	200	89110/178142	−35.95	155
	5	200	157069/314050	−122.3	318

TABLE 3.4: Résumé de l'optimisation pour $g = 1000$ et $\Omega = 3, 3.5, 4$ et 5

	g	M	N_t/N_v	E(u)	Itérations	CPU (s)
IPOPT	5000	200	114830/229568	10.28	615	16658 (4.6h)
	10000	200	144172/288264	14.38	952	44206 (12.2h)
	15000	200	170490/340912	17.67	880	35483 (9.8h)

TABLE 3.5: Résumé de l'optimisation pour le potentiel harmonique, $\Omega = 0.95$ et $g = 5000,\ 10000$ et 15000

Simulations en 3D

Nous avons également mené quelques calculs en dimension trois après optimisation du code. Ces modifications, consistent pour l'essentiel à pré-calculer une unique fois pour chaque maillage la matrice associée aux parties bilinéaires quadratiques de l'énergie et celle associée aux contraintes afin de n'avoir à calculer que des produits matrice-vecteur ou des produits scalaires à chaque appel de fonction pour le calcul de ces termes :

$$E_h(u) = \int_{\mathcal{D}_h} \underbrace{\frac{1}{2}|\nabla u|^2 + V|u|^2 - \frac{1}{2}\mathbf{\Omega} \wedge \mathbf{x}.(iu, \nabla u)}_{\text{partie bilinéaire}} + \frac{g}{2}|u|^4$$

On sacrifie ainsi la mémoire nécessaire au stockage d'une matrice supplémentaire, de la dimension du problème, mais on économise de ce fait une quantité non négligeable d'intégrations sur le volume de calcul. L'avantage en dimension trois est indéniable, d'autant plus que la mémoire, avec l'avènement des processeurs 64 bits, est désormais une ressource dont on dispose en masse.

Autre stratégie incontournable pour éviter les calculs superflus, l'adaptation de maillage joue ici encore un rôle central sans lequel ces simulations n'auraient certainement pas été possibles sur notre humble ordinateur personnel. On utilise pour cela, d'une part le logiciel `mshmet` qui permet de calculer une métrique anisotrope à partir d'un ensemble de fonctions éléments finis 3D (voir chapitre 1), et `mmg3d` qui utilise cette métrique pour raffiner un maillage et l'adapter à ces fonctions.

On se base sur les travaux de Danaila ([35], [36]) pour établir un ensemble de test. Tout d'abord, en essayant de récupérer les formes en U et en S des vortex obtenus dans [35] pour un potentiel harmonique légèrement anisotrope dans les directions x et y :

$$\omega_x = 1. \quad \omega_y = 1.06 \quad \omega_z = 0.067$$

FIGURE 3.15: Condensat en forme de cigare obtenu pour $\frac{\omega_z}{\omega_\perp} = 0.067$

La faible valeur de ω_z relativement aux pulsations associées aux deux autres directions est responsable de la forme très allongée du condensat évoquant celle d'un cigare. La vitesse de rotation $\Omega = 0.5$ est prise proche de celle à partir de laquelle apparaissent

les premiers vortex ($\Omega = 0.42$ d'après [35]) pour éviter la nucléation de plusieurs vortex. Les configurations de vortex en U et S étant des minima locaux, il est nécessaire d'initialiser l'optimisation par un condensat comportant un vortex de forme proche :

$$u_0^U(x,y,z) = \Phi_\alpha(x,y,z)\sqrt{\rho_{\mathrm{TF}}(x,y,z)}$$

Pour le vortex en U, Φ_U doit décrire un vortex d'axe Oz. [35] préconise un léger décalage dans la direction Ox, ce qui donne :

$$\Phi_U(x,y,z) = \sqrt{\frac{1}{2} + \frac{1}{2}\tanh\left[\frac{4}{\epsilon}\left(\sqrt{(x-d)^2+y^2} - \epsilon\right)\right]}\, e^{i\arg[x-d+iy]}$$

Nous avons cependant obtenu un vortex en U en partant d'une configuration symétrique (axiale, $d = 0$).

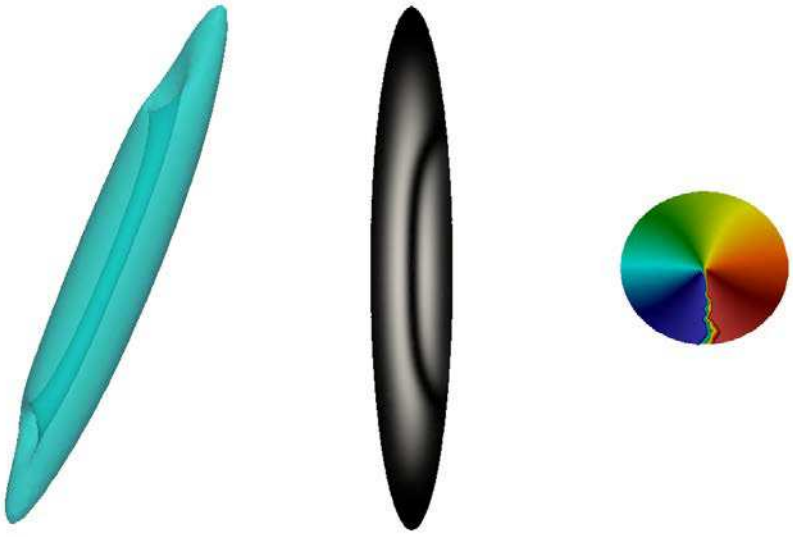


Figure 3.16: Isosurface de faible densité, densité et phase de l'approximation 3D d'un condensat par éléments finis avec IPOPT pour un potentiel harmonique légèrement anisotrope ($\omega_x = 1$, $\omega_y = 1.06$, $\omega_z = 0.067$, $g = 1250$ $\Omega = 0.5$, maillage adapté anisotrope comportant 25905 points).

L'initialisation en vue de l'obtention d'un vortex en S nécessite quant à elle une fonction d'onde dont la densité s'annule artificiellement le long d'une courbe en S. Ceci s'obtient en prenant d fonction de z dans la fonction précédente :

$$\Phi_S(x,y,z) = \sqrt{\frac{1}{2} + \frac{1}{2}\tanh\left[\frac{4}{\epsilon}\left(\sqrt{(x-d(z))^2+y^2} - \epsilon\right)\right]}\, e^{i\arg[x-d(z)+iy]}$$

avec, par exemple :

$$d(z) = \begin{cases} -1 + \dfrac{\tanh\left[\gamma\left(1+\dfrac{z}{h}\right)\right]}{\tanh(\gamma)} & \text{si } z < 0 \\ 1 + \dfrac{\tanh\left[\gamma\left(-1+\dfrac{z}{h}\right)\right]}{\tanh(\gamma)} & \text{si } z \geq 0 \end{cases}$$

Les paramètres γ et h contrôlent respectivement la hauteur et la courbure de la courbe décrite par la zone de densité nulle.

On retrouve effectivement la configuration en U comme minimum par minimisation avec IPOPT et approximation par éléments finis (voir figure 3.16). En revanche, nous n'avons pas réussi à capturer de vortex en S, malgré l'affirmation de leur existence en tant qu'état stable d'énergie non minimale pour toute vitesse de rotation. C'est là un phénomène caractéristique d'IPOPT avec lequel il s'est révélé difficile dans la pratique de capter un minimum local choisi lorsqu'il existe d'autres minima d'énergie inférieure proches.

Nous avons, toujours dans le cas du potentiel harmonique, effectué divers calculs avec une pulsation de piégeage selon z plus proche de celle selon les autres directions, et avec soit des pulsations identiques selon x et y ($\omega_x = \omega_y = 1$), soit une anisotropie franche, mais, et surtout, avec une vitesse proche de la limite de $\Omega = 1$ et une constante d'interaction atomique relativement élevée ($g = 5000$) afin d'observer de multiples vortex et de pouvoir éprouver la fidélité avec laquelle le maillage s'adapte à ces géométries complexes.

FIGURE 3.17: Potentiel quadratique + quartique avec $\Omega = 1.5$ et $g = 2500$

3.4.3 Perspectives

Les résultats sont encourageants, et on obtient des simulations assez précises pour l'identification de formes et de vortex avec des calculs de quelques heures, ce qui dans le cas de la dimension trois est assez rapide. Des surfaces plus régulières pourraient être obtenues si l'on disposait de solveurs linéaires parallèles compatibles avec IPOPT, ce qui n'était pas le cas au moment ou nous avons effectué ces calculs.

Mais avant tout, il est important de relever l'avantage net que permet IPOPT en termes de temps de développement. L'étude [36] se base sur une méthode d'optimisation qu'il a fallu programmer, tester et valider. Ce sont plusieurs mois de travail qui se trouvent en amont de ces résultats restreints à la dimension deux. Le code développé à l'aide de

Figure 3.18: Potentiel harmonique anisotrope avec $\Omega = 0.95$ et $g = 5000$

l'interface IPOPT pour FreeFem++ a quant à lui nécessité une dizaine de jours, validation incluse. L'adaptation du code à la dimension trois n'a demandé que deux semaines supplémentaires, et les résultats sont d'une précision tout à fait convenable.

3.5 Un problème de contact

L'interface pour IPOPT nous a permis de cosigner un article avec Zakaria Belhachmi (Université de Haute Alsace), Faker Ben Belgacem (Université de Compiègne) et Frédéric Hecht qui avaient besoin de résultats numériques pour illustrer une analyse mathématique d'un modèle pour un problème de contact avec conditions aux bords de Signorini. Intitulé *Quadratic finite elements with non-matchin grids for the unilateral boundary contact*, cet article a été publié dans la revue ESAIM : Mathematical Modelling and Numerical Analysis. Nous l'incluons ici dans sa version originale en anglais. Les notations qui y sont utilisées peuvent différer de celles employées jusqu'ici, et ce, jusqu'au chapitre suivant. La référence précise figure quant à elle dans la bibliographie à l'entrée [7].

Nous y contribuons par les résultats numériques apparaissant en fin d'analyse, dans le dernier paragraphe. Il s'agit de vérifier en pratique les vitesses de convergence prévues par la théorie. On utilise pour cela FreeFem++ ainsi que l'interface développée pour IPOPT afin de résoudre numériquement le problème énoncé en section 3.5.5 pour diverses valeurs du paramètre de précision des maillages h, en prenant soin d'avoir choisi des seconds membres tels que l'on connaisse la solution exacte du problème. Selon que l'on utilise ou non le mortier et que les grilles sont conformes ou non, on obtient les courbes des figures 3.22 et 3.23, par comparaison dans différentes normes des solutions exactes à celles calculées par le processus d'optimisation.

Dans tous les cas, les conditions de contact unilatéral pour le problème discrétisé s'expriment naturellement comme des contraintes d'inégalité impliquant linéairement les degrés de liberté localisés sur le bord de contact. La fonctionnelle est quant à elle quadratique, définie sur un espace de relativement grande dimension, puisque les paramètres d'optimisation consistent en chacun des degrés de liberté des espaces de fonctions éléments finis. Que ce soit la hessienne de la fonctionnelle, ou l'application linéaire permettant d'exprimer les contraintes, les matrices rencontrées sont essentiellement creuses. Ces caractéristiques font de ce problème numérique un champ d'application idéal pour la méthode de points intérieurs implémentée dans IPOPT.

En pratique, le code se montre très efficace, avec une convergence systématique vers la

solution exacte en un nombre relativement faible d'itérations, typiquement entre huit et trente selon la finesse du maillage. Cette rapidité et cette stabilité doivent cependant être relativisées. Le problème est en effet construit de sorte à ce que, bien que le minimiseur active les contraintes, il coïncide toutefois avec celui de la fonctionnelle sans contrainte. Il est très difficile, et probablement impossible, d'identifier des seconds membres et fonctions pour les conditions aux bords qui aboutissent à la fois à une solution qui peut être exprimée à l'aide de fonctions élémentaires, et à un minimiseur de la fonctionnelle libre qui violerait strictement les contraintes de contact unilatéral. De ce fait, en menant les calculs effectués pour obtenir les courbes de convergences de cette étude, IPOPT ne se heurtait pas nécessairement à l'intégralité des difficultés susceptibles d'être rencontrées dans la résolution de ce problème. Si on choisit des seconds membres et des conditions aux bords tels que l'on soit certain que les contraintes sont fortement actives, bien que la méthode réussisse toujours à converger, elle va requérir en général un plus grand nombre d'itérations que pour les calculs menés pour l'article qui suit, de l'ordre de vingt pour-cent.

Le code FreeFem++ est relativement long et, ayant été écrit avec une version embryonnaire de l'interface IPOPT qui ne comportait pas toutes les opportunités de simplification de la version actuelle, celui-ci illustre plutôt ce qui pourrait être économisé en termes de complexité de code, que les possibilités actuelles de l'interface pour IPOPT. Le lecteur pourra le trouver en annexe B.4.

3.5.1 Introduction

High-order finite elements are more and more relevant for complex systems since they allow for large computations with increased accuracy. Engineers of unilateral contact may be interested in as an afforded response to capture small structures in their numerical simulations. Letting apart the linear finite element to which a wide literature is devoted. We refer to the books [49, 62, 98] as classical references, then to [20] for earlier a priori error estimates and to [67, 57, 54] for more recent studies which improve former results in the literature. Conducting the convergence analysis of the finite element approximation applied to unilateral problems turns out to be a pain in the neck, presumably because they are expressed by variational inequalities. Unlike the linear variational equations for which a general theory of the finite element methods has been developed, specific tools should be used for the unilateral contact inequality according to the degree of the finite elements. Here, we are interested in the Signorini boundary contact. Some of the mixed finite elements that are performing tools to efficiently handle the numerical locking caused by the nearly incompressible materials in elasticity, are based on quadratic elements (see [21]). One may think for example of Taylor-Hood elements already used for the unilateral contact (see [87, 12, 55]). These all are as many reasons why it is worth to study quadratic finite elements for the unilateral contact. We are involved in non-matching meshes and in the use of the mortar method. Deep attention has been successfully devoted to a single mesh context in Signorini's problem or in the case of compatible grids that is the meshes within each component of the system to handle numerically coincide at the interfaces. Detailed studies with optimal convergence estimates are available for those finite element methods with matching grids (see [10]). When the meshes are constructed independently, resulting in dissimilar grids that do not match at interfaces, a substantial implementation and validation work have been achieved (see [42, 43, 83, 56, 66, 82]) either in unilateral or bilateral contact with or without friction. We recommend in particular the interesting survey provided in [65]. In the numerical analysis chapter, only few attempts have been realized for the unilateral configurations (see [53, 56]). We pursue here a study of some numerical modelings of unilateral contact in the frame of the mortar-matching for finite

elements with degree two, applied to the primal variational Signorini Inequality. A literature exists on the linear mortar finite element method for the Signorini problem. We refer to [11, 51, 52, 95, 9] without being exhaustive. Users are recommended not to enforce the unilateral conditions on the discrete solution before running the mortar projection. Such a mortar matching, first introduced in [14], allows to express the unilateral inequalities, in the discrete level, on traces built on the same grid and avoids by then possible instabilities along interfaces. It is nowadays widely known that writing unilateral inequalities directly on the approximated solution at the interfaces as done in the classical node-to-segment treatments, generates low numerical accuracy. We refer to [43, 63, 97] for a worthy discussion about the efficiency of the mortar method in interface contact mechanics.

Our willing is to focus only on the mortaring and technical issues related to, for conciseness. We exclude therefore any other features which are doubtless as important in unilateral contact and that are either already successfully handled or under consideration in many research teams. We pay all our attention to the mortar projections along interfaces of dissimilar grids, to the way the unilateral contact is enforced at the discrete level and finally to show how the combination of both yields optimal convergence results, owing to Falk's Lemma. The outline of the paper is as follows. Section 3.5.2 is a presentation of some models where unilateral contact is involved. We describe briefly the static problem of the unilateral contact between two elastic bodies, the solution to be computed is the displacement field. We recall also the unilateral conditions in the case of the Laplace equation. The regularity issue for the solution of the Signorini-Laplace problem is discussed at the end of the section. In section 3.5.3, we introduce the mortar framework for the quadratic finite elements and the unilateral conditions are then expressed at the nodes of the interfaces. After recalling a suitable version of Falk's Lemma, we dedicate section 3.5.4 to the numerical analysis of the mortar quadratic finite element method for the Signorini-Laplace problem. The convergence rates we exhibit in our main result, Theorem 3.5.1, are similar to the case where matching grids are used. As a result, using incompatible grids in the simulation of unilateral contact does not degrade the accuracy when mortar-matching drives the communication between different meshes. The final section is dedicated to some numerical experiences to check the theoretical predictions. Lastly, in Section 3.5.5, we describe and comment some experiences realized in `Freefem++` to assess the theoretical predictions proved here.

Some notations.— Let $\mathscr{C}(\Omega)$ be the space of real valued continuous function on a given domain Ω. The Lebesgue space of square integrable functions $L^2(\Omega)$ is endowed with the natural inner product $(\cdot,\cdot)_{L^2(\Omega)}$; the associated norm is $\|\cdot\|_{L^2(\Omega)}$. The Sobolev space $H^1(\Omega)$ involves all the functions that are in $L^2(\Omega)$ so as their first order derivatives. It is provided with the norm $\|\cdot\|_{H^1(\Omega)}$ and the semi-norm is denoted by $|\cdot|_{H^1(\Omega)}$. For any portion of the boundary $\Upsilon \subset \partial\Omega$, the space $H^1_0(\Omega,\Upsilon)$ contains all the functions of $H^1(\Omega)$ that vanish on Υ. We recall that, for any $\theta \in]0,1[$, the Sobolev space $H^\theta(\Upsilon)$ can be obtained by a Hilbertian interpolation between $H^1(\Upsilon)$ and $L^2(\Upsilon)$ and $H^{-\theta}(\Upsilon)$ is the dual space $(H^\theta(\Upsilon))'$. Notice finally that any function $v \in H^1_0(\Omega, \partial\Omega \setminus \Upsilon)$ whose Laplacian is in $L^2(\Omega)$ has a normal derivative $(\partial_{\boldsymbol{n}} v)_{|\Upsilon}$ that belongs to $H^{-1/2}(\Upsilon)$. We refer to [3] for a detailed study of the fractional Sobolev spaces.

3.5.2 Examples of Unilateral Conditions

A large number of thermal, mechanical or electrical devices can be modeled by unilateral boundary conditions. We refer the reader to the book by J. Duvaut and J.-L. Lions

[38] for various examples. In fact the two examples we describe here are picked up from that reference.

Unilateral Contact between two Solids

Let us consider two elastic solids occupying the domains Ω^s and Ω^i in their initial unconstrained configurations. They are assumed in contact with each other along a common part of their boundaries ($\partial\Omega^s \cap \partial\Omega^i$). These boundaries $\partial\Omega^s$ and $\partial\Omega^i$ are supposed regular and the unit external normal is denoted by $\boldsymbol{n}$ when no confusion is feared or $\boldsymbol{n}^i$ and $\boldsymbol{n}^s$ for the corresponding bodies. Each of $\partial\Omega^\ell$ ($\ell = i$ or s) is divided into three portions $\Gamma_D^\ell, \Gamma_N^\ell$ et Γ_C^ℓ (see Figure **??**). The solid Ω^ℓ is fixed along Γ_D^ℓ and is subjected to a traction or compression force $\boldsymbol{g}^\ell (\in \boldsymbol{L}^2(\Gamma_N^\ell))$ on Γ_N^ℓ and to a volume force $\boldsymbol{f}^\ell (\in \boldsymbol{L}^2(\Omega^\ell))$, which is most often its own weight. Both solids share a common zone $\Gamma_C = \Gamma_C^s = \Gamma_C^i$, candidate to an effective contact.

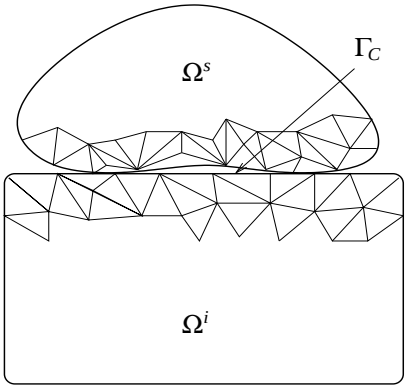


FIGURE 3.19: Two solids are in unilateral contact along Γ_C. Inter-penetration is not authorized.

The unilateral contact problem consists in : *finding a displacement fields* $\boldsymbol{u} = (\boldsymbol{u}^\ell)_\ell = (\boldsymbol{u}_{|\Omega^i}, \boldsymbol{u}_{|\Omega^s})$ *satisfying the boundary value problem*

$$-\boldsymbol{div}\ \sigma^\ell(\boldsymbol{u}^\ell) = \boldsymbol{f}^\ell \qquad \text{in } \Omega^\ell, \tag{3.40}$$

$$\boldsymbol{u}^\ell = 0 \qquad \text{on } \Gamma_D^\ell, \tag{3.41}$$

$$\sigma^\ell(\boldsymbol{u}^\ell)\boldsymbol{n}^\ell = \boldsymbol{g}^\ell \qquad \text{on } \Gamma_N^\ell. \tag{3.42}$$

The bold symbol $\boldsymbol{div}$ denotes the divergence operator of a tensor field and is defined as $\boldsymbol{div}\ \sigma = \left(\frac{\partial \sigma_{kr}}{\partial x_r}\right)^k$. The bodies are made of Hook type material whose constitutive laws are given by $\sigma^\ell(\boldsymbol{u}^\ell) = A^\ell(\boldsymbol{x})\,\epsilon^\ell(\boldsymbol{u}^\ell)$, where $A^\ell(\boldsymbol{x})$ is a fourth order symmetric and elliptic tensor. Frictionless unilateral contact conditions enforced on Γ_C allow to close the system

$$(\sigma^i \boldsymbol{n}^i) \cdot \boldsymbol{n}^i = (\sigma^s \boldsymbol{n}^s) \cdot \boldsymbol{n}^s = \sigma_{\boldsymbol{n}}, \tag{3.43}$$

$$[\boldsymbol{u} \cdot \boldsymbol{n}] \leq 0, \qquad \sigma_{\boldsymbol{n}} \leq 0, \qquad \sigma_{\boldsymbol{n}}[\boldsymbol{u} \cdot \boldsymbol{n}] = 0, \tag{3.44}$$

$$\sigma_{\boldsymbol{t}}^i = \sigma_{\boldsymbol{t}}^s = 0. \tag{3.45}$$

The notation $[\boldsymbol{u} \cdot \boldsymbol{n}] = (\boldsymbol{u}^i \cdot \boldsymbol{n}^i + \boldsymbol{u}^s \cdot \boldsymbol{n}^s)$ stands for the jump of the normal displacements contact across Γ_C. Condition (3.43) expresses the Newton action and reaction principle, (3.45) indicates that the contact occurs without friction. Most often, the modeling of the

contact condition is formulated using a gap function ζ defined on Γ_C, so that instead of $[\boldsymbol{u}\cdot\boldsymbol{n}]\leq 0$ and of the saturation condition $\sigma_{\boldsymbol{n}}[\boldsymbol{u}\cdot\boldsymbol{n}]=0$ we have $[\boldsymbol{u}\cdot\boldsymbol{n}]\leq\zeta$ and $\sigma_{\boldsymbol{n}}([\boldsymbol{u}\cdot\boldsymbol{n}]-\zeta)=0$ on the contact zone Γ_C (see [38]). As the whole subsequent analysis can be extended straightforwardly to the case where ζ does not vanish, we choose, only for conciseness, $\zeta=0$. In what follows we will write σ^ℓ for $\sigma^\ell(\boldsymbol{u}^\ell)$. The analysis is limited to infinitesimal deformations (small perturbations) and therefore to the case of linear elasticity where the strain tensor produced by the displacement $\boldsymbol{v}$ is

$$\epsilon(\boldsymbol{v})=\frac{1}{2}(\nabla\boldsymbol{v}+(\nabla\boldsymbol{v})^T).$$

In order to derive the variational formulation of the unilateral contact problem (3.40)-(3.45) we need the spaces

$$\boldsymbol{X}(\Omega)=\boldsymbol{H}^1_0(\Omega^i,\Gamma^i_D)\times\boldsymbol{H}^1_0(\Omega^s,\Gamma^s_D).$$

This space is endowed with the broken semi-norm : $\forall\boldsymbol{v}\in\boldsymbol{X}(\Omega)$,

$$|\boldsymbol{v}|_{*,H^1}=\Big(\|\epsilon(\boldsymbol{v}^i)\|^2_{\boldsymbol{L}^2(\Omega^i)}+\|\epsilon(\boldsymbol{v}^s)\|^2_{\boldsymbol{L}^2(\Omega^s)}\Big)^{1/2}.$$

which is, because of the Dirichlet condition and Korn's inequality, a norm equivalent to the broken norm $\|.\|_{*,H^1}$. Then, the appropriate closed convex set $\boldsymbol{K}(\Omega)$ of admissible displacements is defined by

$$\boldsymbol{K}(\Omega)=\Big\{\boldsymbol{v}=(\boldsymbol{v}^i,\boldsymbol{v}^s)\in\boldsymbol{X}(\Omega),\qquad([\boldsymbol{v}\cdot\boldsymbol{n}])_{|\Gamma_C}\leq 0\Big\}.$$

The weak formulation results in a variational inequality that reads as : *find* $\boldsymbol{u}\in\boldsymbol{K}(\Omega)$ *such that*

$$a(\boldsymbol{u},\boldsymbol{v}-\boldsymbol{u})\geq F(\boldsymbol{v}-\boldsymbol{u}),\qquad\forall\boldsymbol{v}\in\boldsymbol{K}(\Omega).\tag{3.46}$$

In (3.46), we set : $\forall\boldsymbol{u},\boldsymbol{v}\in\boldsymbol{X}(\Omega)$

$$\begin{aligned}a(\boldsymbol{u},\boldsymbol{v})&=\int_{\Omega^i}A^i(\boldsymbol{x})\,\epsilon(\boldsymbol{u}^i):\epsilon(\boldsymbol{v}^i)\,d\boldsymbol{x}+\int_{\Omega^s}A^s(\boldsymbol{x})\,\epsilon(\boldsymbol{u}^s):\epsilon(\boldsymbol{v}^s)\,d\boldsymbol{x},\\ F(\boldsymbol{v})&=\int_{\Omega^i}\boldsymbol{f}^i\cdot\boldsymbol{v}^i\,d\boldsymbol{x}+\int_{\Gamma^i_{\boldsymbol{g}}}\boldsymbol{g}^i\cdot\boldsymbol{v}^i\,d\Gamma+\int_{\Omega^s}\boldsymbol{f}^s\cdot\boldsymbol{v}^s\,d\boldsymbol{x}+\int_{\Gamma^s_{\boldsymbol{g}}}\boldsymbol{g}^s\cdot\boldsymbol{v}^s\,d\Gamma,\end{aligned}$$

The existence and uniqueness of $\boldsymbol{u}\in\boldsymbol{K}(\Omega)$ which solves problem (3.5) result from Stampacchia's Theorem (see [62], [49]).

The Signorini-Laplace problem

Consider the domain Ω as the union of two non-intersecting components Ω^s and Ω^i. Assume that both sub-domains share a common boundary Γ_C. Then, denote by Γ^s_N and Γ^i_N the remaining parts of the boundaries of Ω^s and Ω^i, respectively. We set $\Gamma_N=\Gamma^s_N\cup\Gamma^i_N$ and $\Gamma^\ell=\Gamma_C\cup\Gamma^\ell_N$ for $\ell=i,s$. Finally, let $\boldsymbol{n}_C$ ($\boldsymbol{n}$ if there is no risk of confusion) be the unit normal to Γ_C oriented from Ω^i toward Ω^s .

Assume now that $g=(g^s,g^i)\in L^2(\Gamma_N)$ and $f\in L^2(\Omega)$. The unilateral contact problem we interested in consists in finding $p=(p^i,p^s)$, satisfying the following equations :

$$\begin{aligned}p^\ell-\text{div}\,(a^\ell\nabla p^\ell)&=f\qquad\text{in }\Omega^\ell,\ell=i,s&(3.47)\\ (a^\ell\partial_{\boldsymbol{n}})p^\ell&=g^\ell\qquad\text{on }\Gamma^\ell_N,\ell=i,s&(3.48)\end{aligned}$$

The coefficient $a = (a^i, a^s)$ lies in $L^\infty(\Omega)$. It is such that $0 < a_* \leq a(\boldsymbol{x}) \leq a^* < \infty, \forall \boldsymbol{x} \in \Omega$. To complete the model, unilateral conditions should be provided along Γ_C. They read as

$$(a^i \partial_{\boldsymbol{n}}) p^i = (a^s \partial_{\boldsymbol{n}}) p^s, \tag{3.49}$$

$$[p] = p^i - p^s \geq 0, \qquad (a^i \partial_{\boldsymbol{n}}) p^i \geq 0, \qquad (a^i \partial_{\boldsymbol{n}}) p^i [p] = 0. \tag{3.50}$$

We aim at a variational framework fitted to the problem (3.47)-(3.50). It is provided by the broken space

$$H^1_*(\Omega) = H^1(\Omega^i) \times H^1(\Omega^s).$$

The broken norm is denoted by $\| \cdot \|_{*,H^1}$. Taking into account the unilateral contact condition on Γ_C in the weak formulation is ensured by incorporating it in the closed convex cone

$$K(\Omega) = \Big\{ q \in H^1_*(\Omega), \qquad ([q])_{|\Gamma_C} = (q^i - q^s)_{|\Gamma_C} \geq 0 \Big\}.$$

The primal variational principle applied to problem (3.47)–(3.50) produces the variational inequality consisting in : *finding* $p \in K(\Omega)$ *such that :*

$$a(p, q - p) \geq F(q - p), \qquad \forall q \in K(\Omega). \tag{3.51}$$

In (3.51), we have set

$$a(p, q) = \int_\Omega pq \; d\boldsymbol{x} + \int_{\Omega^i} a^i (\nabla p)(\nabla q) \; d\boldsymbol{x} + \int_{\Omega^s} a^s (\nabla p)(\nabla q) \; d\boldsymbol{x} \tag{3.52}$$

$$F(v) = \int_\Omega fq \; d\boldsymbol{x} + \langle g \; , \; q \rangle_{1/2,\Gamma}. \tag{3.53}$$

Applying here again Stampacchia's theorem (see [49]), we state that the variational inequality (3.51) is well posed and has only one solution in $p \in K(\Omega)$ that depends continuously on the data (f, g).

Remark 3.5.1. In the variational formulation, the mathematical sense given to conditions *(3.50)* is as follows

$$\langle (a^i \partial_{\boldsymbol{n}}) p \; , \; \mu \rangle_{1/2,\Gamma_C} \geq 0, \qquad \forall \mu \in H^{1/2}(\Gamma_C), \; \mu \geq 0,$$
$$\langle (a^i \partial_{\boldsymbol{n}}) p^i \; , \; [p] \rangle_{1/2,\Gamma_C} = 0.$$

Roughly, the first formula says $(a^i \partial_{\boldsymbol{n}}) p^i \geq 0$ on Γ_C while the second expresses the saturation condition $(a^i \partial_{\boldsymbol{n}}) p^i [p] = 0$ on Γ_C.

Remark 3.5.2. Variational inequality (3.51) can be viewed as the optimality condition of a quadratic minimization problem on the closed convex $K(\Omega)$. Hence, the solution $p \in K(\Omega)$ is the one that satisfies

$$\frac{1}{2} a(p, p) - F(p) = \min_{q \in K(\Omega)} \frac{1}{2} a(q, q) - F(q). \tag{3.54}$$

About Singularities

The numerical analysis of any finite element method applied the unilateral contact problem (3.51) requires the knowledge of the regularity of the solution $p = (p^i, p^s)$. Since the work by Moussaoui and Khodja (see [73]), it is admitted that the unilateral condition may generate some singular behavior at the neighborhood of Γ_C. We assume Γ_C is the

union of regular portions connected to each other through some corners. Some singularities are generated by the angular interfaces and some others are specifically created by the contact condition. We will assume that $a^\ell = 1$, this is the most easy case to look at for the regularity of the Signorini solution.

Let us first look at the portions of the unilateral boundary that are straight-lines. The unilateral contact on Γ_C creates some singularities caused by the points where the contact changes from binding([3]) ($p^s = p^i$) to non-binding ($p^i > p^s$). Let $\boldsymbol{m}$ be one of those points. Considering the global pressure $p = (p^i, p^s)$ at the neighborhood of $\boldsymbol{m}$, we have the transmission conditions $(p^i, \partial_{\boldsymbol{n}} p^i) = (p^s, \partial_{\boldsymbol{n}} p^s)$ along the binding portion of Γ_C. As a result, p is harmonic at the neighborhood of that binding boundary. Things happen as if we were involved in a Poisson equation around $\boldsymbol{m}$ and the non-binding part of Γ_C acts like an insulating crack. The shape of the singularities affecting both (p^s, p^i) are

$$\mathcal{S}^i_k(r,\theta) = \alpha_k\ \varphi(r) r^{k+1/2}\ \cos(k+1/2)\theta, \qquad 0 \le \theta \le \pi,$$
$$\mathcal{S}^s_k(r,\theta) = \alpha_k\ \varphi(r) r^{k+1/2}\ \cos(k+1/2)\theta, \qquad \pi \le \theta \le 2\pi.$$

The polar coordinates (r,θ) are used with origin $\boldsymbol{m}$ and φ is a cut-off function around $\boldsymbol{m}$. The effective contact takes place for $\theta = \pi$ and the pressure p is discontinuous at $\theta = 0(2\pi)$. Writing the unilateral conditions (3.49)-(3.50) at both sides of $\boldsymbol{m}$ produces that

$$\mathcal{S}^i_k(r,0) - \mathcal{S}^s_k(r,2\pi) > 0, \qquad \partial_n S^i_k(r,\pi) = \partial_n S^s_k(r,\pi) \ge 0$$

produces $\alpha_k(1-(-1)^k) > 0$ and $(-1)^{k+1}\alpha_k \ge 0$. This yields that k is necessarily odd and $\alpha_k > 0$. Then only the odd k-singularities are involved in the expansion of p^ℓ,

$$p^\ell(r,\theta) = p^\ell_{reg}(r,\theta) + \sum_k \mathcal{S}^\ell_{2k+1}(r,\theta).$$

We expect therefore that $p^\ell \in H^\sigma(V_{\boldsymbol{m}})$ for any $\sigma < \frac{5}{2}$ and $V_{\boldsymbol{m}}$ is a an open set containing $\boldsymbol{m}$. Furthermore, if the first singularities $(\mathcal{S}^\ell_1)_\ell$ are switched off then the next ones $(\mathcal{S}^\ell_3)_\ell$ command the regularity of (p^i, p^s), at least at the neighborhood of Γ_C and one obtains that $p^\ell \in H^\sigma(V_{\boldsymbol{m}})$ for any $\sigma < \frac{9}{2}$. In fact, the point to be fixed once for all is that the possible values of the limit Sobolev regularity index increases by jumps of length two, it equals to $5/2, 9/2, \cdots$. Now, let us focus on the corners. At a vertex $\boldsymbol{m}$ with an aperture of the angle $\omega \in]0, 2\pi[$([4]), three situations are possible. Either, effective contact holds along both edges of the sector, and the global solution (p^i, p^s) is blind to the interface and then non singularities arise. The second situation is where effective contact takes place along one edge while on the other we have no contact. This configuration is a reminiscence to the upper case of a straight-line. The binding edge has no role to play in the shape of the singularities arising around $\boldsymbol{m}$. Only singularities like those discussed for a straight-line enter the asymptotic expansions of (p^i, p^s). It is noticeable that these first two possibilities for corners do not degrade the regularity of the solution (p^i, p^s), already limited by the changing of the contact conditions along a straight-line. The last case turns to be the worst, and consists in the absence of contact on both sides ($p^i < p^s$). The homogeneous normal derivative of both (p^i, p^s) allows to compute the admissible singularities,

$$\mathcal{S}^i_k(r,\theta) = \eta_k + \alpha_k\ \varphi(r) r^{\frac{k\pi}{\omega}} |\log r|\ \cos\frac{k\pi}{\omega}\theta, \qquad 0 \le \theta \le \omega,$$
$$\mathcal{S}^s_k(r,\theta) = \beta_k\ \varphi(r) r^{\frac{k\pi}{2\pi-\omega}} |\log r|\ \cos\frac{k\pi}{2\pi-\omega}(2\pi-\theta) \qquad \omega \le \theta \le 2\pi.$$

3. We adopt here the terminology from the solid mechanics.
4. ω is the angle within Ω^i. The angle in Ω^s is then $(2\pi - \omega)$.

where $\eta_k > 0$ is sufficiently large to ensure that $p^i > p^s$ within the support of φ (at the neighborhood of the corner $\boldsymbol{m}$). Moreover, the logarithm is there for $\mathcal{S}_k^i$ only when $\frac{k\pi}{\omega}$ is an integer, idem for $\mathcal{S}_k^s$ and $\frac{k\pi}{2\pi-\omega}$. The regularity of the first singular functions $(\mathcal{S}_1^i, \mathcal{S}_1^s)$ will fully decide of the smoothness of (p^i, p^s). This predicts that $p^i \in H^\sigma(V_{\Gamma_C})$ for any $\sigma < 1+\frac{\pi}{\omega}$ and $p^s \in H^\sigma(V_{\boldsymbol{m}})$ for any $\sigma < 1 + \frac{\pi}{2\pi-\omega}$. Anyway, we obtain $p^\ell \in H^{3/2}(V_{\Gamma_C})$. Would these singularities be switched off, there would be a jump on the regularity exponents, since $5/2 < \sigma < 1 + \min(\frac{2\pi}{\omega}, \frac{2\pi}{2\pi-\omega})$ and so on.

3.5.3 The Quadratic Mortar Finite Element Method

We pursue the numerical simulation of the unilateral boundary conditions by non-matching finite element grids. The main advantage is the possible generation of meshes well adapted to the local features of the domain components and to the physical parameters. As said earlier, the convergence rate of the finite element approximation depends on the regularity of the solution p. According the previous regularity discussion it may reasonably happen that p belongs to a more regular space than $H^{3/2}$ or even than $H^{5/2}$, at least near Γ_C. Therefore, the approximation of the variational inequality (3.51) by affine finite elements fails to fully account for that regularity of p (at least at the vicinity of Γ_C). We are indeed limited by H^2. Quadratic finite elements for the Signorini problem have been analyzed in [10] and [53]. Many factors command the accuracy of the discretization such as the smoothness of Γ_C, the regularity of f and g, the way Γ_C is approximated by a given mesh, ..., and the unilateral singularities along Γ_C. Our scrutiny work is exclusively oriented toward the effect of this last point, and we do not pay attention to the other factors. They are all classical and have been considered in many works (see [28, 18]). Assume then that the domain Ω is polygonal so that it can be exactly covered by rectilinear finite elements and that Γ_C is a union of straight-lines.

Towards the construction of the discrete space, we recall some fundamental notions. Let $h = (h_i, h_s)$ be a given pair of real positive numbers that will decay towards zero. With each Ω^ℓ, we associate a regular family of triangulations $\mathcal{T}_h^\ell$, with triangular elements, denoted by κ, whose diameter does not exceed h_ℓ. The unilateral boundary Γ_C inherits two independent meshes $\mathcal{T}_h^{\ell,C}(\ell = i, s)$ from $\mathcal{T}_h^\ell$. The mesh $\mathcal{T}_h^{\ell,C}$ on Γ_C is the set of all the edges of $\kappa \in \mathcal{T}_h^\ell$ on the contact zone. The nodes of $\mathcal{T}_h^{\ell,C}$ are $(\boldsymbol{x}_m^\ell)_{1\le m\le m_*^\ell}$, $(t_m^\ell =]\boldsymbol{x}_m^\ell, \boldsymbol{x}_{m+1}^\ell[)_{1\le m\le m_*^\ell}$ are the edges and the middle nodes of t_m is $\boldsymbol{x}_{m+1/2}^\ell$. Denote by $\mathcal{P}_2$ the space of the polynomials on κ with a global degree ≤ 2. With any κ, we associate the finite set Ξ_κ of the vertices of κ, so that $(\kappa, \mathcal{P}_2, \Xi_\kappa)$ is a finite element of Lagrange type. We define $\Xi^\ell = \bigcup_{\kappa\in\mathcal{T}_h^\ell} \Xi_\kappa$. The finite element space in Ω^ℓ is then

$$X_h(\Omega^\ell) = \Big\{ q_h^\ell \in \mathscr{C}(\overline{\Omega}^\ell), \quad \forall \kappa \in \mathcal{T}_h^\ell, \quad (q_h^\ell)_{|\kappa} \in \mathcal{P}_2 \Big\},$$

and $X_h(\Omega) = X_h(\Omega^i) \times X_h(\Omega^s)$. The unilateral condition (3.50) in a discrete level should be driven carefully. Comparing directly q_h^i and q_h^s, built on different meshes, has to be avoided since it is known to create numerical troubles. It is rather recommended to realize it between q_h^i and some projection of q_h^s on the opposite mesh $\mathcal{T}_h^{i,C}$. We therefore need the traces space

$$W_h^\ell(\Gamma_C) = \Big\{ \varphi_h = (q_h^\ell)_{|\Gamma_C}, \qquad q_h^\ell \in X_h(\Omega^\ell) \Big\},$$

which will be the mortar space for a fixed ℓ. Given that Γ_C is a closed curve there is no need to differentiate the Lagrange Multipliers space and the mortar space as currently made

(see [14]). The mortar projection π^ℓ is defined on $W_h^\ell(\Gamma_C)$ as follows. Let $\psi \in L^2(\Gamma_C)$, we have $\pi^\ell\psi \in W_h^\ell(\Gamma_C)$ and

$$\int_{\Gamma_C} (\psi - \pi^\ell\psi)\chi_h \, d\Gamma = 0 \qquad \forall \chi_h \in W_h^\ell(\Gamma_C). \tag{3.55}$$

The operator π^ℓ is an orthogonal projection with respect to the norm of $L^2(\Gamma_C)$ and is therefore expected to satisfy an optimal error estimate with respect to the norm of that space. It is also mandatory that it yields an optimal estimate for the norm of $H^{1/2}(\Gamma_C)$. We hence add a mild assumption on the mesh $\mathcal{T}_h^{\ell,C}$ introduced in [34] : two arbitrary edges t_m^ℓ and $t_{m'}^\ell$ satisfy

$$\frac{|t_m^\ell|}{|t_{m'}^\ell|} \leq \eta\alpha^{|m-m'|}, \qquad (1 \leq m, m' \leq m_*^\ell),$$

where $1 \leq \alpha < 9$ and η does not depend on h_s. $\mathcal{T}_h^{\ell,C}$ is then called an (M)-mesh([5]). As a result, the following stability holds : $\forall \psi \in H^{1/2}(\Gamma_C)$,

$$\|\pi^\ell\psi\|_{H^{1/2}(\Gamma_C)} \leq C\|\psi\|_{H^{1/2}(\Gamma_C)}. \tag{3.56}$$

The proof may be issued from the results in [34, 86] and an interpolation artifice of linear operators in Hilbert spaces. Moreover, π^ℓ satisfies some approximation results. Let $\mu \in [0, 3[$, the following approximation holds : $\forall \psi \in H^\mu(\Gamma_C)$,

$$\|\psi - \pi^\ell\psi\|_{H^{-1/2}(\Gamma_C)} + h_\ell\|\psi - \pi^\ell\psi\|_{H^{1/2}(\Gamma_C)} \leq C(h_\ell)^{\mu+1/2}\|\psi\|_{H^\mu(\Gamma_C)}. \tag{3.57}$$

We proceed with the construction of the discrete convex cone. We have then to decide which side of Γ_C plays the role of the mortar, let us say Ω^i, the same Γ_C as a part of $\partial\Omega^s$ is hence the non-mortar. The purpose consists in enforcing the non-negativity on the values of $(\pi^s(q_h^i) - q_h^s)$ at the vertices of Γ_C coming from the Ω^s side. The discrete closed convex cone is hence defined to be

$$K_h(\Omega) = \Big\{ q_h = (q_h^s, q_h^i) \in X_h(\Omega), \qquad [q_h]_h(\boldsymbol{x}) = (\pi^s(q_h^i) - q_h^s)(\boldsymbol{x}) \geq 0, \quad \forall \boldsymbol{x} \in \Xi^s \cap \Gamma_C \Big\}.$$

The overall tools useful for the discrete variational inequality are available. The Ritz-Galerkin approximation reads as follows : *find* $p_h \in K_h(\Omega)$ *such that*

$$a(p_h, q_h - p_h) \geq F(q_h - p_h), \qquad \forall q_h \in K_h(\Omega). \tag{3.58}$$

The set $K_h(\Omega)$ is obviously an external approximation of $K(\Omega)$, i.e. $K_h(\Omega) \not\subset K(\Omega)$, the mortar discretization is of course non-conforming. Checking that the discrete problem (3.58) has only one solution $p_h \in K_h(\Omega)$ is an easy matter from Stampacchia's Theorem.

Remark 3.5.3. It is possible to split Γ_C into more than one mortar *(resp. non-mortar)*, each edge of Ω^i *(resp.* Ω^s*)* may be considered as mortar *(resp. non-mortar)* by itself. The matching projection has to be particularized to the non mortars, whose union forms Γ_C as part of Ω^s. Each of these projections should be defined in the specific mortaring way *(see [14, 96])*. The test functions ψ_h involved in the definition of *[14]* has to be reshaped as depicted in *Figure 3.20*. Nevertheless, for the issue under examination, the theory for the mortar method may be conducted similarly. We do not therefore consider this case in details.

5. (M)-meshes authorizes the adaptativity contrary to the quasi-uniform meshes.

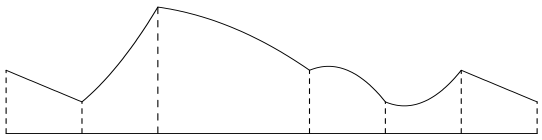


FIGURE 3.20: A test functions ψ_h. Observe that its affine at both extreme edges.

3.5.4 Convergence of the Mortar Method

The analysis of the accuracy of the approximation, as said earlier, will be specifically dedicated to the effect of unilateral condition, excluding the other factors such as the regularity of the data that have been widely investigated in the literature. Henceforth, f and g are taken regular. Moreover, for seek of simplicity we fix the parameter a to unity, that is $a^i = a^s = 1$. The forthcoming results are extended to the general case provided that a is smooth enough.

Relying on the discussion in Section 3.5.2, we will pay a particular attention to the case where the regularity exponent of the solution p is either $3/2$ or $5/2$. The proofs for $\sigma = 3/2$ can be reproduced, with a minor adaptation, when the regularity exponent σ lies in $[1, 3/2[$ and those developed for $\sigma = 5/2$ can be extended to the case where $\sigma \in]2, 5/2[$ following the same lines. In the subsequent the generic constant C depends on the solution $p = (p^i, p^s)$. Our main result in this paper, that we state here and prove in several steps, is the following.

Theorem 3.5.1. Let $p = (p^i, p^s) \in K(\Omega)$ be the solution of the variational inequality *(3.51)*. Assume that $p^\ell \in H^{3/2}(\Omega^\ell)$. Then we have that

$$\|p - p_h\|_{*,H^1} \leq C[(h_i)^{1/2} + (h_s)^{1/2}].$$

Assume that $p^\ell \in H^{5/2}(\Omega^\ell)$. Then we have that

$$\|p - p_h\|_{*,H^1} \leq C[(h_i)^{3/2} + (h_s)^{3/2}].$$

The constant C is dependent on $\|p^\ell\|_{H^\sigma(\Omega^\ell)}$.

Remark 3.5.4. The analysis intermediary case $\sigma \in]3/2, 2]$ requires some more work. The important fact to point at is the necessity here of a technical assumption on the effective contact region that seems to be firstly introduced in *[20]*. The effective contact portion of Γ_C is supposed to be the union of a finite number of segments. In addition, some Sobolev and Sobolev Murray embeddings combined to finite element approximation results Lebesgue spaces $L^r (r > 2)$ are needed in the proof.

Remark 3.5.5. Notice that the issue of optimal convergence rates for higher regularity that is $\sigma \in]5/2, 3]$ is still open. This has nothing to do with the mortaring process. Such results are also missing for the Signorini problem where only a single mesh is involved.

Falk's Lemma in the Mortar Context

Deriving an estimate of the error $(p - p_h)$ with respect to the energy norm relies on a basic tool obtained from the Falk Lemma (see [41], [11]).

Lemma 3.5.2. Let $p \in K(\Omega)$ be the solution of the variational inequality (3.51) and $p_h \in K_h(\Omega)$ be the solution of the discrete variational inequality (3.58). Then we have that

$$\|p-p_h\|^2_{*,H^1} \leq C\Big\{ \inf_{q_h \in K_h(\Omega)} (\|p-q_h\|^2_{*,H^1} \quad + \quad \langle \partial_{\boldsymbol{n}} p^i \ , \ [q_h] \rangle_{1/2,\Gamma_C}) \\ + \quad \inf_{q \in K(\Omega)} \langle \partial_{\boldsymbol{n}} p^i \ , \ [q-p_h] \rangle_{1/2,\Gamma_C} \Big\}. \quad (3.59)$$

Remark 3.5.6. The last infimum is the consistency error, it is the "variational crime" and is due to the nonconformity of the approximation.

The Best approximation error

We pursue an optimal bound of the first infimum in the abstract Falk's estimate (3.59). The following holds.

Proposition 3.5.3. Assume that $p^\ell \in H^\sigma(\Omega^\ell)$ with $\sigma = 3/2$ or $5/2$. Then we have that

$$\inf_{q_h \in K_h(\Omega)} (\|p-q_h\|^2_{*,H^1} + \langle \partial_{\boldsymbol{n}} p^i \ , \ [q_h] \rangle_{1/2,\Gamma_C}) \leq C\big[(h_i)^{\sigma-1} + (h_s)^{\sigma-1}\big]^2. \quad (3.60)$$

Proof. Observe first that $[p] = (p^i - p^s) \in H^{\sigma-1/2}(\Gamma_C)$. Using the lifting theorem (see [3]) we may find $r \in H^\sigma(\Omega^s)$ such that $r_{|\Gamma_C} = [p] \geq 0$ verifying the stability

$$\|r\|_{H^\sigma(\Omega^s)} \leq C\big[\|p^i\|_{H^\sigma(\Omega^i)} + \|p^s\|_{H^\sigma(\Omega^s)}\big]. \quad (3.61)$$

Setting $(\tilde{p}^i, \tilde{p}^s) = (p^i, p^s + r) \in H^\sigma(\Omega^i) \times H^\sigma(\Omega^s)$, it is clear that $[\tilde{p}] = 0$ on Γ_C and thus $\tilde{p} \in H^1(\Omega^i \cup \Omega^s)$. We are therefore in the bilateral contact context. Proceeding as in [11] (see also [14]) we are able to build $(\tilde{q}^i_h, \tilde{q}^s_h) \in X_h(\Omega^i) \times X_h(\Omega^s)$ satisfying the mortar matching that is $[\tilde{q}_h]_h = 0$. Moreover, thanks to (3.56) there holds that

$$\|p - \tilde{q}_h\|_{*,H^1} \leq C\big[(h_i)^{\sigma-1} + (h_s)^{\sigma-1}\big]. \quad (3.62)$$

Now, let $r_h \in X_h(\Omega^s)$ be the Lagrange interpolant of r and define q_h as follows

$$q_h = (q^i_h, q^s_h) = (\tilde{q}^i_h, \tilde{q}^s_h - r_h). \quad (3.63)$$

We check easily that, for all $\boldsymbol{x} \in \Xi^s \cap \Gamma_C$, we have

$$[q_h]_h(\boldsymbol{x}) = r_h(\boldsymbol{x}) = r(\boldsymbol{x}) = [p](\boldsymbol{x}) \geq 0.$$

As a result, we have that $q_h \in K_h(\Omega)$ and the following bound holds

$$\|p-q_h\|_{*,H^1} \leq \|p-\tilde{q}_h\|_{*,H^1} + \|r-r_h\|_{H^1(\Omega^s)} \leq C\|p-\tilde{q}_h\|_{*,H^1} + C(h_s)^{\sigma-1}\|r\|_{H^\sigma(\Omega^s)}$$

Using (3.62) together with (3.61) yields

$$\|p - \tilde{q}_h\|_{*,H^1} \leq C\big[(h_i)^{\sigma-1} + (h_s)^{\sigma-1}\big].$$

The first part of the infimum (3.60) is bounded in an optimal way. There remains to estimate $\langle \partial_{\boldsymbol{n}} p^i \ , \ [q_h] \rangle_{1/2,\Gamma_C}$ which is pretty longer. Furthermore, different technicalities are required according to the regularity exponent σ. We choose to provide the proofs into two separate lemmas that come just after. Once these are stated, the proof of the proposition will be completed. □

Lemma 3.5.4. Assume that $p^\ell \in H^{3/2}(\Omega^\ell)$ and q_h given as in *(3.63)*. Then we have that

$$\langle \partial_{\boldsymbol{n}} p^i \ , \ [q_h] \rangle_{1/2,\Gamma_C} \leq C\big[(h_i)^{1/2} + (h_s)^{1/2}\big]^2.$$

Proof. From the saturation $(\partial_{\boldsymbol{n}} p^i)[p] = 0$ on Γ_C, we derive that

$$\begin{aligned} \langle \partial_{\boldsymbol{n}} p^i \ , \ [q_h] \rangle_{1/2,\Gamma_C} &= \langle \partial_{\boldsymbol{n}} p^i \ , \ [\tilde{q}_h] \rangle_{1/2,\Gamma_C} + \langle \partial_{\boldsymbol{n}} p^i \ , r_h \rangle_{1/2,\Gamma_C} \\ &= \langle \partial_{\boldsymbol{n}} p^i \ , \ [\tilde{q}_h] \rangle_{1/2,\Gamma_C} + \langle \partial_{\boldsymbol{n}} p^i \ , (r_h - r) \rangle_{1/2,\Gamma_C}. \end{aligned} \tag{3.64}$$

The first term is handled in a standard way (see [14, 11]). There remains to bound the second one. That $p^i \in H^{3/2}(\Omega^i)$ with $(-\Delta)p^i \in L^2(\Omega^i)$ results in $(\partial_{\boldsymbol{n}} p^i) \in L^2(\Gamma_C)$ (see [3]). We can then write that

$$\begin{aligned} \langle \partial_{\boldsymbol{n}} p^i \ , (r_h - r) \rangle_{1/2,\Gamma_C} &= \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)(r_h - r) \ d\Gamma \leq \|\partial_{\boldsymbol{n}} p^i\|_{L^2(\Gamma_C)} \|r - r_h\|_{L^2(\Gamma_C)} \\ &\leq C(h_s) \|\partial_{\boldsymbol{n}} p^i\|_{L^2(\Gamma_C)} \|r\|_{H^1(\Gamma_C)} \leq C(h_s) \|p^i\|_{H^{3/2}(\Omega^i)} \|r\|_{H^{3/2}(\Omega^s)}. \end{aligned}$$

Using (3.61) with $\sigma = 3/2$, completes the proof. □

Lemma 3.5.5. Assume that $p^\ell \in H^{5/2}(\Omega^\ell)$ and q_h given as in *(3.63)*. Then we have that

$$\langle \partial_{\boldsymbol{n}} p^i \ , \ [q_h] \rangle_{1/2,\Gamma_C} \leq C\big[(h_i)^{3/2} + (h_s)^{3/2}\big]^2.$$

Proof. The proof is a slightly more complicated than in the previous case. Again, the point is to bound the second term of (3.64). Notice that, due to the regularity of p^ℓ, we obtain

$$\begin{aligned} \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)(r_h - r) \ d\Gamma &= \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)(r_h - r) \ d\Gamma \\ &= \sum_{i=1}^{m^*} \int_{t_m} (\partial_{\boldsymbol{n}} p^i)(r - r_h) \ d\Gamma \leq \sum_{i=1}^{m^*} \|\partial_{\boldsymbol{n}} p^i\|_{L^2(t_m)} \|r - r_h\|_{L^2(t_m)}. \end{aligned}$$

Some indices in the sum may be removed. Are kept there only the indices $m \in \tilde{M}$ for which $(\partial_{\boldsymbol{n}} p^i)_{|\Gamma_C} \subset H^1(\Gamma_C) \subset \mathscr{C}(\Gamma_C)$ vanishes at least once in t_m. Otherwise if $(\partial_{\boldsymbol{n}} p^i)_{|t_m} > 0$ then $r_{|t_m} = [p]_{|t_m} = 0$. This gives $(r_h)_{|t_m} = 0$ and the associated integral is zero. Then, the inequality reduces to

$$\int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)(r_h - r) \ d\Gamma \leq C \sum_{m \in \tilde{M}} \|\partial_{\boldsymbol{n}} p^i\|_{L^2(t_m)} (h_s)^2 |r|_{H^2(t_m)}$$

Applying now Lemma 8.1 of [10] to $\partial_{\boldsymbol{n}} p^i \in H^1(\Gamma_C)$, we obtain

$$\begin{aligned} \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)(r_h - r) \ d\Gamma &\leq \sum_{m \in \tilde{M}} C(h_i) |\partial_{\boldsymbol{n}} p^i|_{H^1(t_m)} (h_s)^2 |r|_{H^2(t_m)} \\ &\leq C(h_i)(h_s)^2 \Big(\sum_{m \in \tilde{M}} |\partial_{\boldsymbol{n}} p^i|^2_{H^1(t_m)} \Big)^{1/2} \Big(\sum_{m \in \tilde{M}} |r|^2_{H^2(t_m)} \Big)^{1/2} \\ &\leq C(h_i)(h_s)^2 |\partial_{\boldsymbol{n}} p^i|_{H^1(\Gamma_C)} |r|_{H^2(\Gamma_C)}. \end{aligned}$$

The proof is achieved due to the estimate (3.61) with $\sigma = 5/2$. □

The Consistency Error

Now, we are left with the consistency error. The analysis of this error requires some sharp technicalities.

Lemma 3.5.6. Assume that $p^\ell \in H^{3/2}(\Omega^\ell)$. Then we have that

$$\inf_{q\in K(\Omega)} \langle \partial_{\boldsymbol{n}} p^i \ , \ [q-p_h]\rangle_{1/2,\Gamma_C} \leq C\big[(h_s)^{1/2}\|p-p_h\|_{*,H^1} + (h_s)\big].$$

Proof. The proof takes four (arborescent) steps, though not that difficult. Given that $(\partial_{\boldsymbol{n}} p^i) \in L^2(\Gamma_C)$, taking $q = p$ yields that

$$\langle \partial_{\boldsymbol{n}} p^i \ , \ [p-p_h]\rangle_{1/2,\Gamma_C} = \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)[p-p_h]\ d\Gamma.$$

Plugging $[p-p_h]_h$ results in

$$\langle \partial_{\boldsymbol{n}} p^i \ , \ [p-p_h]\rangle_{1/2,\Gamma_C} = \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)[p-p_h]_h\ d\Gamma - \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)((I-\pi^s)(p^i-p_h^i))\ d\Gamma. \quad (3.65)$$

Both integrals involved in there are handled separately. We start by the second integral because it is shorter.
(*i.*) By the Cauchy-Schwarz inequality we have that

$$\int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)((I-\pi^s)(p^i-p_h^i))\ d\Gamma \leq \|\partial_{\boldsymbol{n}} p^i\|_{L^2(\Gamma_C)}\|(I-\pi^s)(p^i-p_h^i)\|_{L^2(\Gamma_C)}.$$

Owing to (3.57), we obtain that

$$\int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)((I-\pi^s)(p^i-p_h^i))\ d\Gamma \leq C(h_s)^{1/2}\|p^i-p_h^i\|_{H^{1/2}(\Gamma_C)} \leq C(h_s)^{1/2}\|p-p_h\|_{*,H^1}.$$

(*ii.*) Now, we turn to estimating the first integral in the bound of (3.65). Let χ_h be the orthogonal projection of $(\partial_{\boldsymbol{n}} p^i)$ on the piecewise constant functions built on the mesh $\mathcal{T}_h^{s,C}$. Actually $((\chi_h)_{|t_m})_m$ are obtained as the average values of $(\partial_{\boldsymbol{n}} p^i)$ on $(t_m)_m$ and are then non-negative. Additionally we have the following bound (see [8]),

$$\|(\partial_{\boldsymbol{n}} p^i) - \chi_h\|_{H^{-1/2}(\Gamma_C)} \leq C(h_s)^{1/2}\|\partial_{\boldsymbol{n}} p^i\|_{L^2(\Gamma_C)} \leq C(h_s)^{1/2}\|p^i\|_{H^{3/2}(\Omega^i)}.$$

The derivation of the desired bound is based on the following decomposition

$$\int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)[p-p_h]_h\ d\Gamma = \int_{\Gamma_C} ((\partial_{\boldsymbol{n}} p^i) - \chi_h)[p-p_h]_h\ d\Gamma + \int_{\Gamma_C} \chi_h[p-p_h]_h\ d\Gamma.$$

Processing in the same lines as above we come up with the estimation

$$\int_{\Gamma_C} ((\partial_{\boldsymbol{n}} p^i) - \chi_h)[p-p_h]_h\ d\Gamma \leq C(h_s)^{1/2}\|p-p_h\|_{*,H^1}. \quad (3.66)$$

In the other side, using the Simpson formula and given that $\chi_h \geq 0$, and $[p_h]_h \geq 0$, it is possible to get rid of p_h to obtain that

$$\int_{\Gamma_C} \chi_h[p-p_h]_h\ d\Gamma \leq \int_{\Gamma_C} \chi_h[p]_h\ d\Gamma,$$

or again that

$$\int_{\Gamma_C} \chi_h [p - p_h]_h \, d\Gamma \leq \int_{\Gamma_C} \chi_h [p] \, d\Gamma + \int_{\Gamma_C} \chi_h (p^i - \pi^s(p^i)) \, d\Gamma.$$

Let us, here, look at each bound independently.

(*iii.*) Using Cauchy-Schwarz' inequality we have that

$$\begin{aligned}\int_{\Gamma_C} \chi_h ((p^i - \pi^s(p^i)) \, d\Gamma &\leq \|\chi_h\|_{L^2(\Gamma_C)} \|p^i - \pi^s(p^i)\|_{L^2(\Gamma_C)} \\ &\leq C \|\partial_{\boldsymbol{n}} p^i\|_{L^2(\Gamma_C)} (h_s) \|p^i\|_{H^1(\Gamma_C)} \leq C h_s.\end{aligned}$$

(*iv.*) The bound of the remaining term is processed as follows. We need η_h the orthogonal projection of $[p]$ on the piecewise constant functions built on the mesh $\mathcal{T}_h^{s,C}$. The unilateral conditions and that χ_h is the orthogonal projection of $(\partial_{\boldsymbol{n}} p^i)$ allows to write

$$\begin{aligned}\int_{\Gamma_C} \chi_h [p] \, d\Gamma &= \int_{\Gamma_C} (\chi_h - (\partial_{\boldsymbol{n}} p^i))[p] \, d\Gamma \\ &= \int_{\Gamma_C} (\chi_h - (\partial_{\boldsymbol{n}} p^i))([p] - \eta_h) \, d\Gamma = - \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)([p] - \eta_h) \, d\Gamma.\end{aligned}$$

Another use of the Cauchy-Schwarz' inequality yields that

$$\int_{\Gamma_C} \chi_h [p] \, d\Gamma \leq \|\partial_{\boldsymbol{n}} p^i\|_{L^2(\Gamma_C)} \|[p] - \eta_h\|_{L^2(\Gamma_C)} \leq C \|(\partial_{\boldsymbol{n}} p^i)\|_{L^2(\Gamma_C)} (h_i) \|[p]\|_{H^1(\Gamma_C)} \leq C(h_i).$$

The proof is achieved by aggregating the bounds of (*i.*), (*iii.*) and (*iv.*). □

Before handling the case where $p^\ell \in H^{5/2}(\Omega^\ell)$, we need a preparatory lemma. Let t be a segment with length $h = |t|$. The following holds.

Lemma 3.5.7. For any $\alpha \in]\frac{3}{2}, 2]$, there exists a constant $C > 0$ so that : $\forall \psi \in H^\alpha(t), \forall a \in t$,

$$\|\psi(x) - (\psi(a) + \psi'(a)(x - a))\|_{L^2(t)} \leq C h^\alpha |\psi|_{H^\alpha(t)}.$$

The constant C does not depend on a.

Proof. Consider for a while the reference segment $\hat{t} = (0, 1)$. Then the Sobolev space $H^\alpha(\hat{t})$ is continuously embedded in the space $\mathscr{C}^1(\hat{t})$ (see [3]). Let $\hat{\psi} \in H^\alpha(\hat{t})$. Set $\hat{\lambda}(\hat{x}) = \hat{\psi}(\hat{x}) - (c + d\hat{x})$. It is straightforward that $\hat{\lambda} \in H^\alpha(\hat{t})$ and we have

$$\sup_{\hat{x} \in \hat{t}} |\hat{\lambda}(\hat{x}) - (\hat{\lambda}(\hat{a}) + \hat{\lambda}'(\hat{a})(\hat{x} - \hat{a}))| \leq \hat{c} \|\hat{\lambda}\|_{H^\alpha(\hat{t})},$$

or again that

$$\sup_{\hat{x} \in \hat{t}} |\hat{\psi}(\hat{x}) - (\hat{\psi}(\hat{a}) + \hat{\psi}'(\hat{a})(\hat{x} - \hat{a}))| \leq \hat{c} \|\hat{\psi} - (c\hat{x} + d)\|_{H^\alpha(\hat{t})}.$$

Since (c, d) are arbitrary, calling for the Bramble-Hilbert Theorem we have

$$\sup_{\hat{x} \in \hat{t}} |\hat{\psi}(\hat{x}) - (\hat{\psi}(\hat{a}) + \hat{\psi}'(\hat{a})(\hat{x} - a))| \leq \hat{c} \inf_{c,d \in \mathbb{R}} \|\hat{\psi} - (c + d\hat{x})\|_{H^\alpha(\hat{t})} \leq \hat{c} |\hat{\psi}|_{H^\alpha(\hat{t})}.$$

Then, we derive that

$$\|\hat{\psi}(\hat{x}) - (\hat{\psi}(\hat{a}) + \hat{\psi}'(\hat{a})(\hat{x} - \hat{a}))\|_{L^2(\hat{t})} \leq \|\hat{\psi}(\hat{x}) - (\hat{\psi}(\hat{a}) + \hat{\psi}'(\hat{a})(\hat{x} - \hat{a}))\|_{L^\infty(\hat{t})} \leq \hat{c} |\hat{\psi}|_{H^\alpha(\hat{t})}.$$

A scaling transformation $\hat{x} \mapsto x$ and $\psi(x) = \hat{\psi}(\hat{x})$ provide the lemma with $C = \hat{c}$. □

Lemma 3.5.8. Assume that $p^\ell \in H^{5/2}(\Omega^\ell)$. Then we have that

$$\inf_{q\in K(\Omega)} \langle \partial_{\boldsymbol{n}} p^i \ , \ [q-p_h]\rangle_{1/2,\Gamma_C} \leq C\big[(h_i)^{3/2}\|p-p_h\|_{*,H^1} + (h_s)^3\big].$$

Proof. The structure of the proof is similar to the previous lemma, though some modifications are necessary.

(*i.*) Setting $\psi_h = \pi^s(\partial_{\boldsymbol{n}} p^i)$, the duality allows to write that

$$\begin{aligned}\int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)\big((I-\pi^s)(p^i-p_h^i)\big)\,d\Gamma &= \int_{\Gamma_C} \big((\partial_{\boldsymbol{n}} p^i)-\psi_h\big)\big((I-\pi^s)(p^i-p_h^i)\big)\,d\Gamma \\ &\leq \|(\partial_{\boldsymbol{n}} p^i)-\psi_h\|_{H^{-1/2}(\Gamma_C)}\|(I-\pi^s)(p^i-p_h^i)\|_{H^{1/2}(\Gamma_C)}.\end{aligned}$$

Owing to (3.56) and (3.57), we obtain that

$$\int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)\big((I-\pi^s)(p^i-p_h^i)\big)\,d\Gamma \leq C(h_s)^{3/2}\|p^i-p_h^i\|_{H^{1/2}(\Gamma_C)} \leq C(h_s)^{3/2}\|p-p_h\|_{*,H^1}.$$

(*ii.*) The estimate on $((\partial_{\boldsymbol{n}} p^i)-\chi_h)$ becomes

$$\|(\partial_{\boldsymbol{n}} p^i)-\chi_h\|_{H^{-1/2}(\Gamma_C)} \leq C(h_s)^{3/2}\|p^i\|_{H^{5/2}(\Omega^i)}.$$

(*iii.*) We have that (ψ_h is defined in (*i.*))

$$\begin{aligned}\int_{\Gamma_C} \chi_h\big((p^i-\pi^s(p^i))\,d\Gamma &= \int_{\Gamma_C} \big(\chi_h-(\partial_{\boldsymbol{n}} p^i)\big)\big(p^i-\pi^s(p^i)\big)\,d\Gamma + \int_{\Gamma_C} (\partial_{\boldsymbol{n}} p^i)\big(p^i-\pi^s(p^i)\big)\,d\Gamma \\ &= \int_{\Gamma_C} \big(\chi_h-(\partial_{\boldsymbol{n}} p^i)\big)\big(p^i-\pi^s(p^i)\big)\,d\Gamma + \int_{\Gamma_C} \big((\partial_{\boldsymbol{n}} p^i)-\psi_h\big)\big(p^i-\pi^s(p^i)\big)\,d\Gamma.\end{aligned}$$

Using Cauchy-Schwarz' inequality yields that

$$\begin{aligned}\int_{\Gamma_C} \chi_h\big((p^i-\pi^s(p^i))\,d\Gamma &\leq \big(\|\chi_h-(\partial_{\boldsymbol{n}} p^i)\|_{L^2(\Gamma_C)} + \|(\partial_{\boldsymbol{n}} p^i)-\psi_h\|_{L^2(\Gamma_C)}\big)\|p^i-\pi^s(p^i)\|_{L^2(\Gamma_C)} \\ &\leq C(h_s)\|(\partial_{\boldsymbol{n}} p^i)\|_{H^1(\Gamma_C)}(h_s)^2\|p^i\|_{H^2(\Gamma_C)} \leq C(h_s)^3.\end{aligned}$$

(*iv.*) The unilateral conditions allow to write

$$\int_{\Gamma_C} \chi_h[p]\,d\Gamma = \int_{\Gamma_C} \big(\chi_h-(\partial_{\boldsymbol{n}} p^i)\big)[p]\,d\Gamma \leq \sum_{i=1}^{m_*^s} \|\chi_h-(\partial_{\boldsymbol{n}} p^i)\|_{L^2(t_m)}\|[p]\|_{L^2(t_m)}.$$

Arguing like in the proof of Lemma 3.5.5, some indices in the sum are canceled. Remains there only the indices $m \in \tilde{M}$ for which $[p]$ vanishes at least once inside t_m. Otherwise if $[p] > 0$ then $(\partial_{\boldsymbol{n}} p)_{|t_m} = 0$ and so does $(\chi_h)_{|t_m}$. This results in

$$\int_{\Gamma_C} \chi_h[p]\,d\Gamma = \int_{\Gamma_C} \big(\chi_h-(\partial_{\boldsymbol{n}} p^i)\big)[p]\,d\Gamma \leq C(h_s)\sum_{i\in\tilde{M}} |\partial_{\boldsymbol{n}} p^i|_{H^1(t_m)}\|[p]\|_{L^2(t_m)}.$$

Now, let us have a close look at $[p] \in \mathscr{C}^1(\Gamma_C)$ restricted to t_m with $m \in \tilde{M}$. It vanishes at least once inside t_m, say at a. Given that $[p] \geq 0$, then a realizes the minimum value of $[p]$. As a result, we obtain that $[p]'(a) = 0$. Next, applying Lemma 3.5.7 to $[p]_{|t_m}$ with $\alpha = 2$, it comes out

$$\begin{aligned}\int_{\Gamma_C} \chi_h[p]\,d\Gamma &\leq C(h_s)\sum_{i\in\tilde{M}} |(\partial_{\boldsymbol{n}} p^i)|_{H^1(t_m)}\|[p]\|_{L^2(t_m)} \\ &\leq C(h_s)^3\sum_{i\in\tilde{M}} |(\partial_{\boldsymbol{n}} p^i)|_{H^1(t_m)}|[p]|_{H^2(t_m)} \leq C(h_s)^3|\partial_{\boldsymbol{n}} p^i|_{H^1(\Gamma_C)}|[p]|_{H^2(\Gamma_C)}.\end{aligned}$$

The proof is completed after putting together the bounds of (*i.*), (*iii.*) and (*iv.*). □

Global estimations

We are now well equipped to provide the expected optimal estimate on the error $(p-p_h)$ generated by the mortar quadratic finite element approximation.

Proof of Theorem 3.5.1: Here σ may equal $3/2$ or $5/2$. Assembling the estimate by Proposition 3.5.3 and Lemma 3.5.6, for $\sigma = 3/2$, or Lemma 3.5.8, for $\sigma = 5/2$, produces

$$\|p - p_h\|^2_{*,H^1} \leq C\Big(\big[(h_i)^{\sigma-1} + (h_s)^{\sigma-1}\big]^2 + (h_i)^{\sigma-1}\|p - p_h\|_{*,H^1}\Big).$$

The proof is complete by Young's inequality.

■

3.5.5 Numerical Tests

We describe some indicative and informative numerical experiences for the Signorini-Laplace problem. The particular aim is to quantify the mortar capacity to preserve the accuracy of quadratic finite elements and to illustrate the agreement between the convergence rate observed in the computations and the one predicted by the theory. Computations for two examples are carried out by means of the code `freefem++` developed by F. Hecht and his coworkers (see [79]). We recall that the computational reliability and robustness of the mortar method in the mechanical unilateral contact has already been proved in many work (see [65] and the references therein)

In both examples, the domains are rectangular, $\Omega^s = [-1,1]\times[-1,0]$ and $\Omega^i = [-1,1]\times[0,1]$. The hypothetic contact zone Γ_C is the common edge $[-1,1]\times\{0\}$, Along the left (vertical) edges $\{1\}\times[-1,0]$ and $\{1\}\times[0,1]$, Neumann conditions are prescribed and along the remaining four edges we enforce Dirichlet boundary conditions. The exact solution for the first example, represented in Figure 3.21, is given by

$$\begin{aligned} p^i(x_1,x_2) &= x_1x_2 + \big[2\max(x_1,0) - \min(x_1,0)\sin(2\pi x_2)\big]\sin(3x_1),\\ p^s(x_1,x_2) &= x_1x_2 + \big[-2\max(x_1,0) - \min(x_1,0)\sin(2\pi x_2)\big]\sin(3x_1). \end{aligned}$$

Notice that the unilateral contact conditions are rather expressed as follows

$$[p] \geq 0, \qquad \partial_{\boldsymbol{n}} p^i \geq \xi, \qquad (\partial_{\boldsymbol{n}} p - \xi)[p] = 0 \qquad \text{on } \Gamma_C.$$

The gap function is $\xi(x_1) = -x_1$. The foregoing analysis remains of course valid as it is and provides the same convergence rates.

Once the discrete finite element objects are constructed, the unilateral inequality is handled as a minimization problem as formulated in (3.54). The unilateral contact conditions on the degrees of freedom located on Γ_C are written under inequality constraints. The obtained optimization problem can therefore be solved iteratively by the interior point strategy, also called the barrier method (see [16]).

Now, to assess the mortar method, combined to the quadratic finite elements, we compare the results it provides to those obtained by two other procedures. The first one is related to the simple interpolation process, used on the same incompatible meshes, for the unilateral conditions. This means that the contact is expressed in a discrete level without practicing the mortar projection π^s. It is therefore changed to

$$[q_h]_h(\boldsymbol{x}) = (q_h^i - q_h^s)(\boldsymbol{x}) \geq 0, \quad \forall \boldsymbol{x} \in \Xi^s \cap \Gamma_C.$$

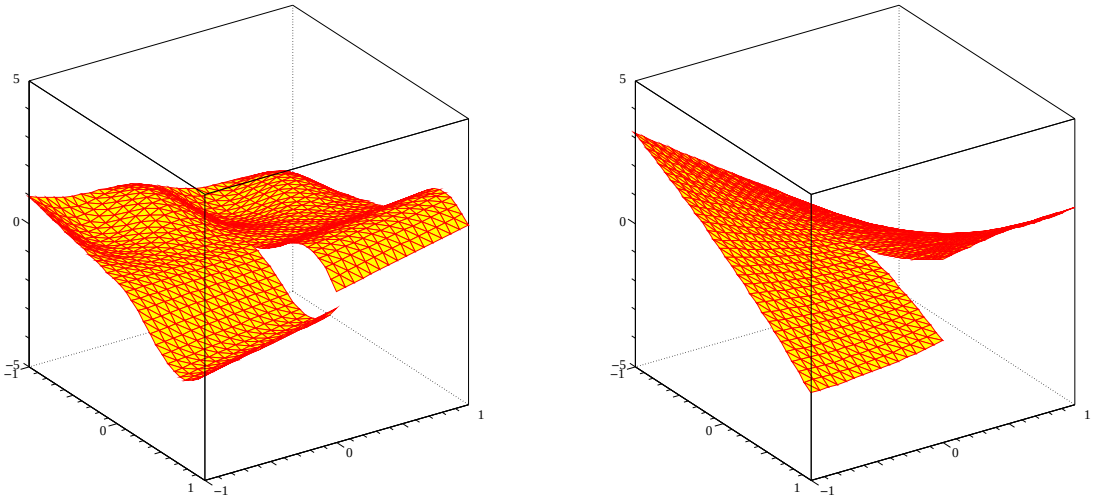


Figure 3.21: The Mortar Finite Element Signorini solutions.

The second approach, the mortar method is compared to, consists in considering conforming meshes with almost the same size as those used in the mortaring process. Consequently, no special projection is needed to glue the discretizations at the contact region. By the way, the mortar projection is nothing else than the identity.

Before discussing the convergence curves, let us figure out the smoothness of $p = (p^i, p^\ell)$. p^i and p^ℓ contain, each, a singularity at the origin $\boldsymbol{m} = (0,0)$. Indeed, $\boldsymbol{m}$ splits Γ_C into an effective contact zone ($x_1 \leq 0$) and a non-contact ($x_1 \geq 0$) region. The singular effect remains limited to Γ_C and is not spread in the interior of the domains. Our aim is to focus on what happen along Γ_C while avoiding contamination by any other inaccuracy factors. The Sobolev regularity of both functions p^i and p^ℓ is therefore limited to $(5/2)_-$([6]). According to Theorem 3.5.1, the convergence rate, with respect to the broken H^1-norm, should be close to $3/2$. In fact, we expect that limitation to be effective when the singular point $\boldsymbol{m}$ lies within an edge of a triangle. Otherwise, if it coincides with a vertex of some triangles, the computations may show some super-convergence result.

In all the simulations, the meshes are triangular and locally structured. We start by running computations when the singular point $\boldsymbol{m}$ is located at the middle of a triangular edge. The H^1, L^2 and L^∞-convergence curves, for the conforming , the non-conforming-mortar and the non-conforming-interpolation methods are provided in Figure 3.22, in logarithmic scales. Obtaining the convergence rates of each method passes by the evaluation of the slopes of the error curves. Let us focus for a while on the H^1-convergence for which we dispose of theoretical estimates at least for the conforming and non-conforming mortar approximations. The linear regressions of the H^1-errors produce slopes that are close to 1.82 and 1.85, respectively. This is slightly better than the value predicted by the analysis which is equal to 1.5. In addition, mortaring the matching enjoys an accuracy level close to that given by conforming grids. Oppositely, the slope 1.19, for the approximation based on the crude interpolation for non-matching grids, suggests that it behaves like a linear and not as a quadratic method. Actually, the results appear satisfactory for meshes of moderate sizes. Things start to worsen for fine meshes where the behavior of the in-

6. This means that $p^\ell \in H^\sigma(\Omega^\ell)$ for all $\sigma < 5/2$.

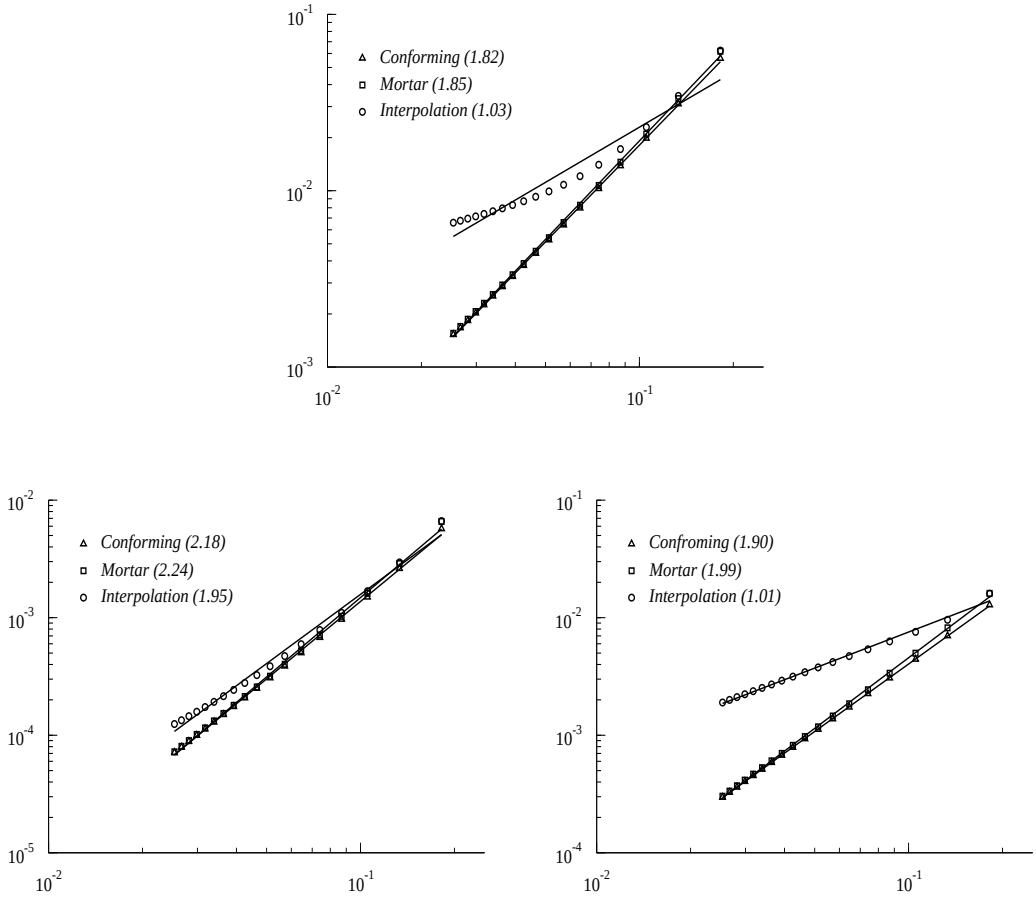


FIGURE 3.22: H^1- (top), L^2- (left-bottom) and L^∞- (right-bottom) convergence curves. The slopes of the linear regressions are provided between parenthesis in the legend.

terpolation matching suffers from some weakness and confirms the well known grievances about this way to proceed. A glance to the right-bottom panel in Figure 3.22 allows similar conclusions for the convergence rates with respect to the L^∞-norm. The L^2-convergence curves show that the conforming and the mortar methods have similar behaviors, and the convergence rate is not that far from 2.5. In fact, after a careful examination of those curves, we found out that the slopes are slightly sagged down when the mesh size gets smaller. The evaluation of the slope of the first half of the curves result in the values 2.41 for the conforming discretization and 2.53 for the mortar method. The explanation of the sag (for the complete curves) is that the error inherent to the optimization solver grows up with the size of the discrete problem. It may then pollute the finite element error, the global accuracy of the approximation is hence affected and the convergence rate is slowed down.

If the singularity support is located at a vertex of some triangles then the quadratic finite element approximation may not see the full singular behavior of the solution. It remains partially blind to the jump of the second derivative of the solution with respect to x_1 at the singular point. Super-convergence may hence show up for both conforming and mortar approximations. The convergence rates with respect to the H^1-norm are almost equal to 2, and are close to 3 for the L^2-norm. The difficulties with the interpolation

matching for non-conforming grids to enjoy similar behavior are substantially aggravated. It has to be avoided, especially for methods with order higher than one. This is already widely known for the bilateral contact models. Curves in Figure 3.23 confirm it for the unilateral contact problems.

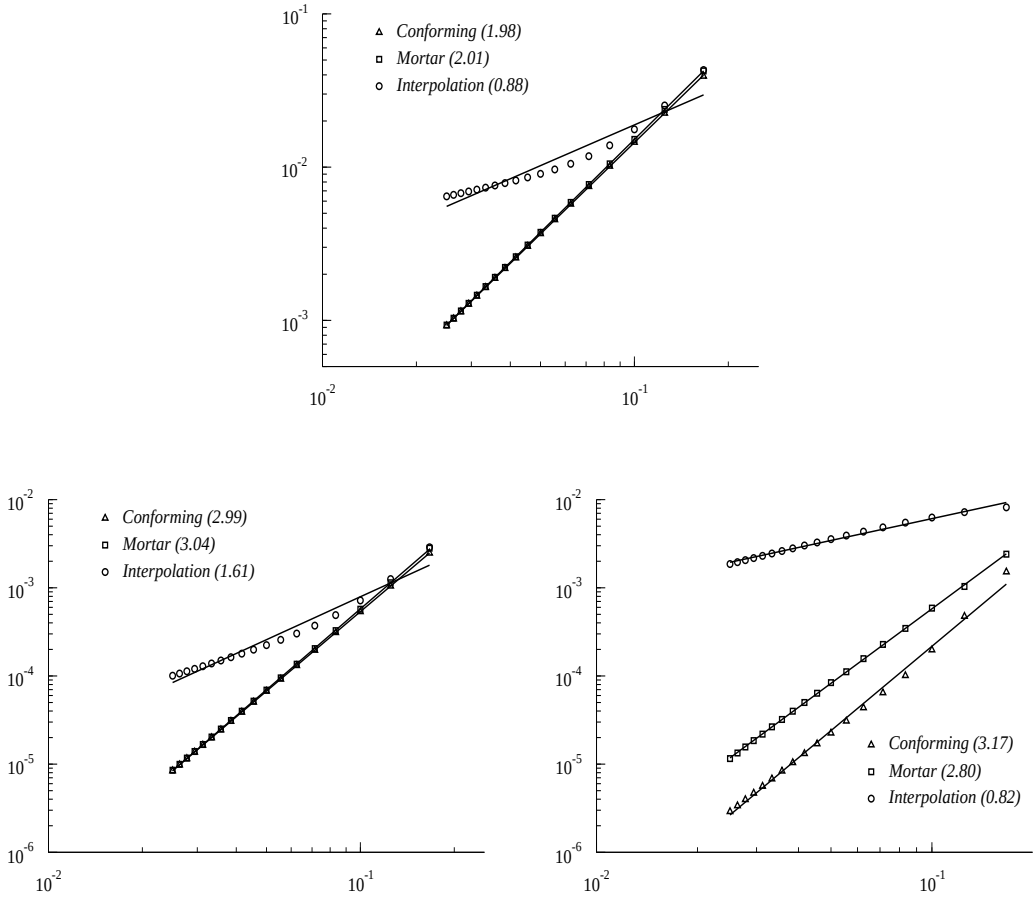


Figure 3.23: Super Convergence phenomenon for some particular meshes.

To close with the first example, let us draw the attention of practitioners to the way the mortar matching should be handled and implemented. The computed solution turns out to be sensitive to the accuracy of the quadrature formula used to evaluate the integrals concerning the matching formula, which is strongly impacted on the construction of the mortar projection. We resume both calculations related to the 'normal' and 'super'-convergence rates. In Figure 3.24, are depicted the H^1-convergence curves when the mortar integrals are exactly computed and when an integration Gauss formula is used with two Gauss nodes within each edge. The plot where super-convergence is expected displays a clear degradation of the accuracy, especially for fine grids, caused by the numerical integration in the mortar matching. To learn more about this issue, we recommend the interesting references [24, 40].

The second test deals with a solution affected by the natural singularity born at the point splitting Γ_C into an activated contact and non-contact zones. The exact solution is

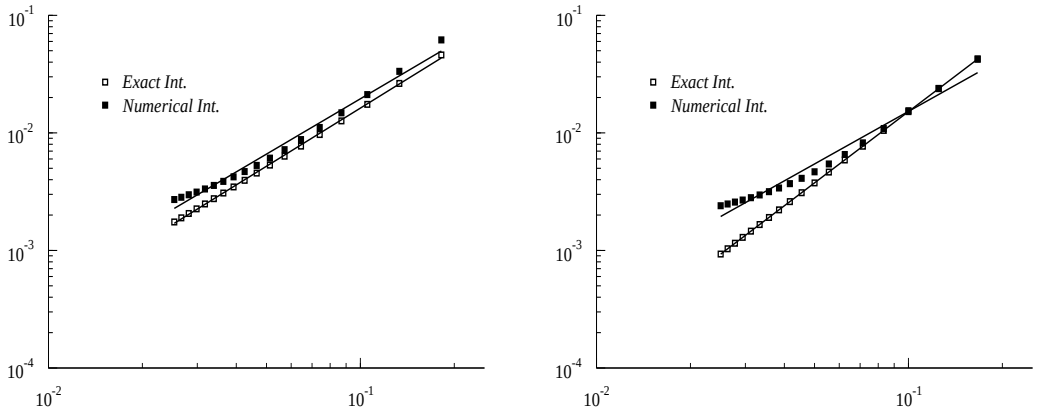


FIGURE 3.24: Effect of the numerical integration on the H^1-converge. The curves to the right are related to the super-convergent case.

defined to be

$$p^i(x_1, x_2) = x_1 x_2 + x\sqrt{r + x} - y\sqrt{r - x},$$
$$p^s(x_1, x_2) = x_1 x_2 - x\sqrt{r + x} - y\sqrt{r - x},$$

where $r = \sqrt{x^2 + y^2}$. It is depicted in the right panel of Figure 3.21. The non-smooth part of the solution looks like the first singularity $r^{3/2}\cos(3\theta/2)$ switched on by the unilateral contact at the vicinity of the separation point $\boldsymbol{m}$ located here again at the origin. We refer to the subsection 3.5.2 the discussion about the singularities. The solution p^ℓ belongs to $H^\sigma(\Omega^\ell)$ for any $\sigma < \frac{5}{2}$. This indicates why the convergence rate of the quadratic finite elements should be close to $h^{3/2}$ with respect to the energy norm. This is confirmed by the first diagram of Figure 3.25, for the mortar and conforming approximations. The interpolation matching on non-conforming provides substantially less accurate results. Notice that, unlike the first example, the circular shape of the singularity contained in the solution affects the accuracy not only at the contact edge Γ_C but also in the interior of the domains Ω^i and Ω^s. No super-convergence phenomenon is therefore expected and the numerical results confirms this fact. There is no particular speeding of the convergence when the singular point coincides with the vertex of some triangles. Convergence curves for the L^2 and L^∞ norms are also represented in Figure 3.25. The trends observed in the first example are confirmed. In the unilateral Signorini contact, the mortar matching does not lower the convergence speed of the quadratic finite element approximation realized on non-conforming grids. In the contrary, the point-wise or the interpolation matching has to be thrown away in such a context as it yields substantially less accurate computed solutions. The distribution of the errors due to each approximation are plotted in Figure 3.26. The shape of these errors illustrates that, for the mortar matching, the singularity is the main cause of the limitation in the accuracy while in the interpolation matching, the way the matching is realized seems to be the principle contributor to the lost of the accuracy.

3.5.6 Conclusion

The numerical analysis of the quadratic finite element method with non-matching grids when applied to unilateral contact equations is achieved here. We use the mortar approach

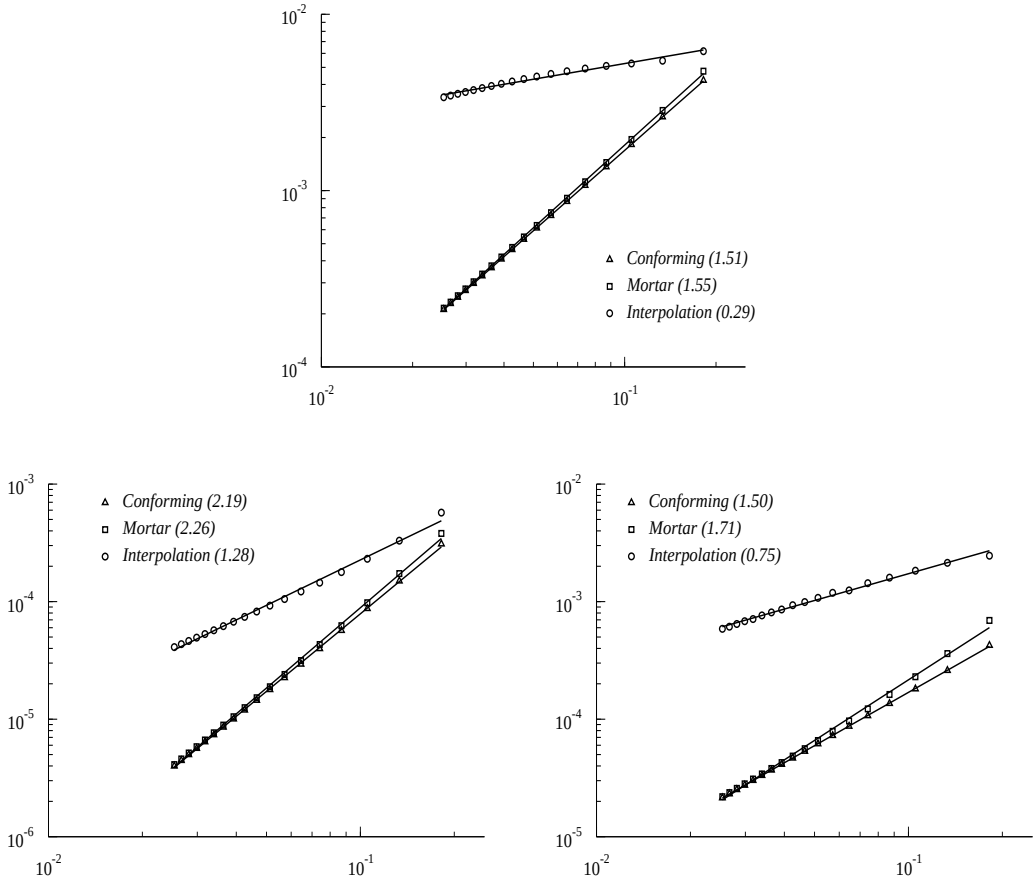


Figure 3.25: Convergence curves for the unilateral singular solution

to drive the communication between different (incompatible) meshes. This mortar concept has been successfully extended to the variational inequalities in [11] for linear finite elements and employed in many computations ; we refer for instance to [52, 9, 42, 63, 83, 82, 97]. The convergence rates established in this paper for the quadratic finite elements are similar to those already stated in [10, 53, 55] when matching grids are employed. We provide some computational examples to illustrate and support the analysis. Let us mention before ending that the mortar quadratic finite element method has already been used for the computational simulation of the unilateral contact in the elasticity (see [43, 56, 27, 83]).

acknowledgement

The authors are indebted to anonymous reviewers whose remarks and comments substantially improves the readability of the paper. Merci to them.

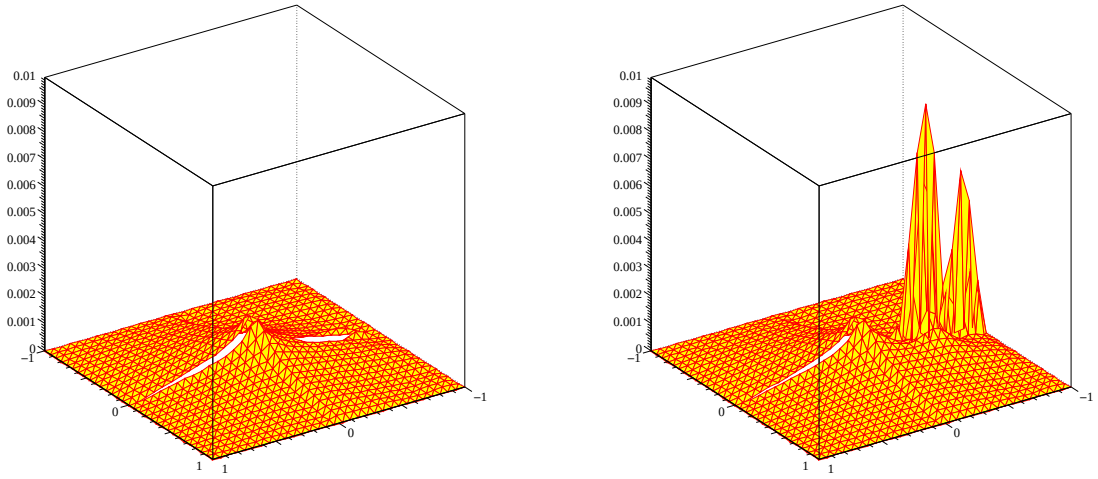


FIGURE 3.26: Error function for the mortar matching (left) where the singularity seems to be the main contributor to the error. The error for the interpolation matching (right) shows that high part of the error is located at the vicinity of the effective contact line, and is necessarily due to the weakness of the point-wise matching.

Chapitre 4

Différentiation Automatique

Ce chapitre est consacré à la seconde des tâches majeures effectuées au cours cette thèse : développer des outils destinés à simplifier les calculs de dérivées dans FreeFem++. Ce gros travail de programmation très technique a occupé une grande partie de ces années de labeur, l'écriture de codes pouvant se révéler incroyablement chronophage. Nous commencerons par donner un exposé des principes essentiels de la différentiation automatique, le but n'étant pas de couvrir le sujet dans son intégralité mais seulement de donner au lecteur les éléments nécessaires à la compréhension de la section suivante, rapportant quant à elle les résultats de la tentative d'implémentation dans FreeFem++. Le lecteur désireux d'approfondir le sujet pourra consulter les ouvrages très détaillés [46] et [76], ainsi que le site [2] qui répertorie les publications ayant un rapport avec la différentiation automatique et la plupart des bibliothèques d'outils de différentiation automatique.

4.1 La différentiation automatique : un aperçu théorique

Le terme *différentiation automatique* désigne un ensemble de techniques permettant l'évaluation des dérivées d'une fonction définie par un programme informatique. Certains membres actifs de la communauté travaillant sur ce sujet préfèrent parler de *différentiation algorithmique* ([46], [76]). Nous n'utiliserons pas cette nomenclature, afin d'éviter une éventuelle confusion avec la différentiation symbolique. Les idées à la base de toutes ces méthodes ne sont plus tout à fait nouvelles et ont déjà été largement appliquées, implémentées et perfectionnées. Ces travaux ont abouti à deux modes majeurs de différentiation automatique qui se déclinent à leur tour en de nombreuses variantes permettant de calculer à la précision machine des dérivées d'ordre quelconque, que ce soit le gradient d'une fonction objectif à optimiser, la jacobienne d'un ensemble de contraintes, une matrice hessienne, ou encore les coefficients jusqu'à un ordre donné d'un développement en série de Taylor. Dans tous les cas, l'idée directrice rassemblant ces techniques est que tout programme implémentant une fonction numérique applique simplement une séquence de fonctions ou d'opérations élémentaires par compositions successives, fonctions que l'on sait différentier de manière exacte. De ce point de vue, différentier la fonction implémentée revient donc à appliquer la formule de dérivation des fonctions composées. Le domaine reste aujourd'hui relativement vivant, les méthodes de différentiation étant encore largement sous-optimales en terme de complexité algorithmique (voir 4.1.5), les travaux actuels dans ce domaine s'efforcent de réduire le coût tant en espace qu'en temps de celles-ci.

Tel qu'énoncé précédemment, le principe sous-jacent à ces méthodes de différentiation *automatique* semble *a priori* n'être ni plus ni moins que celui de la différentiation *symbolique*, utilisée dans les logiciels de calcul formel. Il y a en réalité différence par le choix des

structures algorithmiques auxquelles sont appliquées les règles de dérivation comme nous le verrons plus loin. Nous commencerons donc, avant d'aller plus en profondeur dans le domaine de la différentiation automatique, par donner un bref aperçu des autres méthodes dont dispose le numéricien pour le calcul des dérivées, afin de bien comprendre comment celles-ci se distinguent de la différentiation *automatique* et quels en sont les avantages ou les faiblesses.

4.1.1 Les méthodes alternatives

Si un programme implémente une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ donnée, la solution la plus naturelle pour disposer d'un autre programme évaluant les dérivées de f est, si possible, d'écrire un programme implémentant sa différentielle $df : \mathbb{R}^n \to \mathcal{L}(\mathbb{R}^n, \mathbb{R}^m)$. La remarque est triviale, mais appelle pourtant quelques questions intéressantes. En effet, il est rare que l'on soit amené à implémenter une fonction analytique dont on peut calculer à la main la fonction dérivée. Aussi, dans la plupart des cas, si l'outil informatique doit être utilisé, c'est pour l'implémentation de fonctions fort complexes. En analyse numérique, il s'agira par exemple d'une application f_h issue de la discrétisation d'un modèle mathématique continu f. L'analyse mathématique permettra dans certains cas d'obtenir une différentielle du problème continu df, qui pourra alors être à son tour discrétisée en une fonction $(df)_h$ que l'on implémentera. Or, qu'en est-il de la dérivée du modèle discrétisé $d(f_h)$? Converge-t-elle vers quelque chose de satisfaisant lorsque h devient petit ? Y a-t-il consistance avec la différentielle continue ? Ces questions ne trouvent de réponses qu'au cas par cas, et il sera même parfois plus pertinent d'utiliser la différentielle du modèle discrétisé, afin de prendre en compte certains phénomènes numériques introduits par la méthode de discrétisation qu'il pourrait s'avérer préjudiciable de négliger. Remarquons à ce propos que les différentiations *automatique* et *symbolique* permettent d'obtenir une implémentation de $d(f_h)$ (il ne s'agit donc pas *a priori* d'une discrétisation de la différentielle continue), exacte à la précision machine près quand la méthode des différences finies en donne une approximation $d(f_h)_\epsilon$, et introduisent donc une source d'erreur supplémentaire. Enfin, pour en finir avec ces considérations, remarquons qu'un code dédié à l'évaluation des dérivés et optimisé pour cette tâche sera en général plus performant qu'une implémentation obtenue par l'application d'un algorithme.

Différences finies

Il s'agit d'une méthode bien connue qui consiste, après avoir choisi un paramètre réel $h \neq 0$, à approcher au premier ordre la dérivée partielle de $f : \mathbb{R}^n \to \mathbb{R}^m$ par rapport à sa i-ème variable par :

$$D_i^{+h} f(x) = \frac{f(x + he_i) - f(x)}{h} \tag{4.1}$$

En pratique, on utilisera plus souvent l'opérateur de différences finies *centrées*, précis à l'ordre 2 :

$$D_i^{\pm h} f(x) = \frac{f(x + he_i) - f(x - he_i)}{2h} \tag{4.2}$$

Si cette méthode présente l'énorme avantage de pouvoir utiliser directement une implémentation de f *en boîte noire* (c'est-à-dire sans avoir accès aux détails de son implémentation) par simples appels successifs du programme, elle possède toutefois quelques inconvénients.

Il s'agit tout d'abord de formules approchées (dont l'ordre de précision n'est d'ailleurs valable que pour des fonctions respectivement C^1 et C^2). Si, en théorie, l'approximation est d'autant plus fine que le paramètre h est petit, le choix de h dans la pratique se

révèle plus délicat puisque, en précision finie, la sélection d'un trop petit pas engendrera des erreurs de troncature (on pourra même obtenir des dérivées identiquement nulles si les mantisses de h et x_i sont disjointes !). La valeur optimale de h est *a priori* difficile à évaluer, celle-ci dépendant, en plus de la précision machine ϵ, de la fonction f, de son implémentation et du point en lequel est tentée l'évaluation de la dérivée. Une heuristique visant à s'appliquer dans la majorité des cas est proposée dans [80], où il est suggéré de prendre $h = \sqrt{\epsilon} x_i$.

Pour se convaincre que cette valeur ne fait pas l'hunanimité, examinons la fonction $f : x \mapsto \left|\cos(1) - \frac{\sin(1+x)-\sin(1-x)}{2x}\right|$ pour de petites valeurs positives de x. La figure 4.1 représente le graphe en échelle logarithmique sur les deux axes de l'évaluation machine en double précision de f. En précision infinie, au voisinage de 0, on devrait avoir $f \sim \frac{1}{6}\cos(1)x^2$, comportement que l'on observe sur la portion monotone croissante qui apparaît sur la partie droite de la figure, partie qui correspond donc à l'erreur mathématique commise sur le calcul par différences finies de la dérivée de la fonction sinus en 1. On y constate graphiquement le comportement quadratique de l'erreur au voisinage de 0. La partie fortement oscillante mais globalement décroissante avec la pente -1 correspond aux valeurs de x pour lesquelles l'erreur de troncature est plus importante que l'erreur mathématique. Quant à la petite plage de valeurs de x sur laquelle f paraît constante à la valeur $\cos(1)$, il s'agit de la zone dans laquelle x est si petit devant 1 que l'ordinateur approche $1 \pm x$ par 1. On identifie dans ce cas une valeur optimale du paramètre de différences finies située autour de 10^{-5}. Utiliser la valeur 10^{-8} proposée par [80] mène dans ce cas à la perte de trois chiffres significatifs sur l'approximation effectuée.

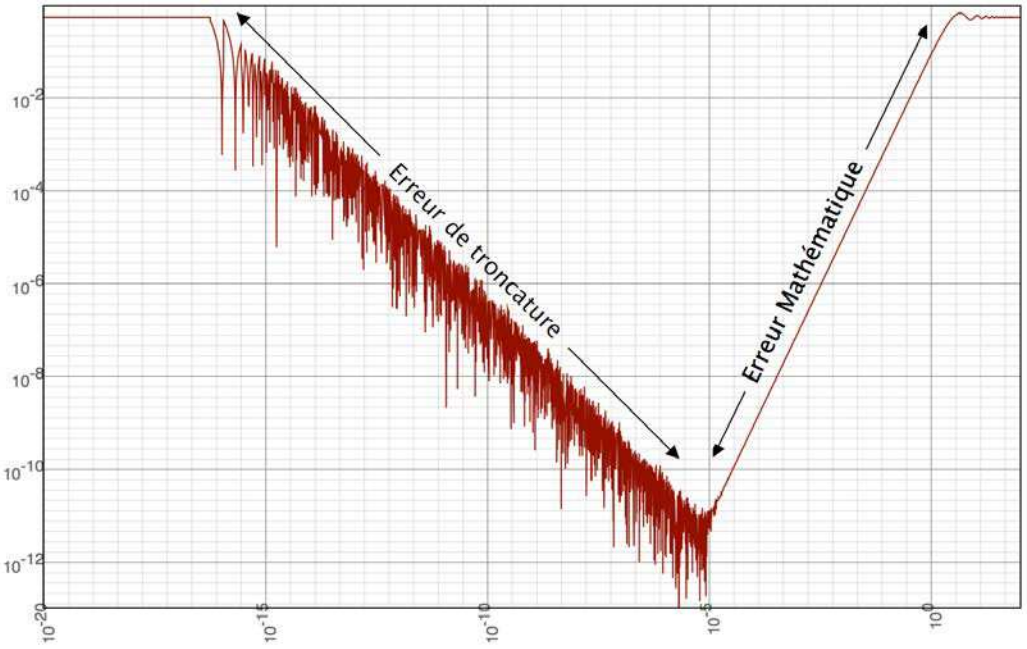


Figure 4.1: Graphe log-log de l'approximation en double précision de $x \mapsto \left|\cos(1) - \frac{\sin(1+x)-\sin(1-x)}{2x}\right|$ sur l'intervalle $[0, 1]$

Plus généralement, on peut montrer que, en plus de dépendre de la fonction différentiée et de x_0, la valeur optimale du pas de différences finies varie avec la racine cubique de la précision machine, et non avec sa racine carrée. On considère pour cela une fonction

$f : I \subset \mathbb{R} \to \mathbb{R}$ et $x_0 \in I$ tel que f admette une dérivée à l'ordre trois en x_0. On définit alors sur un voisinage de x_0 la fonction :

$$\begin{array}{rccc} g & : & V(x_0) & \longrightarrow & \mathbb{R} \\ & & h & \longmapsto & \left| f'(x_0) - \frac{f(x_0+h)-f(x_0-h)}{2h} \right| \end{array} \tag{4.3}$$

Le graphe en double échelle logarithmique d'une approximation numérique en double précision de g aura la même allure que 4.1. C'est-à-dire qu'il comportera une composante constante à la valeur $|f'(x_0)|$ sur l'intervalle $[0, \epsilon x_0]$ correspondant à la zone de non recouvrement des mantisses des représentations numériques de x_0 et h, une partie oscillante mais globalement décroissante en $1/h$ sur l'intervalle $[\epsilon x_0, h_{\text{opt}}]$ révélatrice du gain en précision à mesure que les erreurs de troncature se réduisent, et enfin une partie monotone à croissance quadratique sur $[h_{\text{opt}}, +\infty] \cap V(x_0)$. On peut expliciter g sur ce dernier intervalle, un développement limité à l'ordre trois aboutissant à $g(h) \underset{h\to 0}{\sim} \frac{h^2}{6} \left| f^{(3)}(x_0) \right|$. La valeur optimale du pas de différences finies vérifie donc : $\dfrac{\epsilon x_0 |f'(x_0)|}{h_{\text{opt}}} = \dfrac{h_{\text{opt}}^2}{6} \left| f^{(3)}(x_0) \right|$, dont on déduit, en supposant que $f^{(3)}$ ne s'annule pas en x_0 :

$$h_{\text{opt}} \simeq \sqrt[3]{\frac{6\epsilon x_0 |f'(x_0)|}{|f^{(3)}(x_0)|}} \tag{4.4}$$

Si $f^{(3)}$ s'annule en x_0 il faut étendre le développement limité à l'ordre cinq, ou plutôt l'écrire jusqu'au premier ordre impair n en lequel la dérivée correspondante est non nulle. On réécrirait alors l'équation ci-dessus avec le membre de droite $g(h) \underset{h\to 0}{\sim} \frac{h^{n-1}}{n!} \left| f^{(n)}(x_0) \right|$. La pente de la partie monotone du graphe log-log de g serait dans ce cas d'autant plus grande que l'est n, avec apparition d'une seconde zone constante à la valeur ϵ pour les fonctions dont assez de dérivées s'annulent en x_0. Le pas optimal serait ici $h_{\text{opt}} = \left(\frac{n! \epsilon x_0 |f'(x_0)|}{|f^{(n)}(x_0)|} \right)^{1/n}$.

Le cas où toutes les dérivées d'ordre impair s'annulent en x_0 peut se rencontrer lorsque la fonction à différentier est un polynôme de degré deux. On peut théoriquement utiliser n'importe quelle valeur de h, mais en pratique, seront toujours présentes des erreurs de troncature et, dans ce cas, à moins d'être confronté à un polynôme aux coefficients pathologiques, le pas optimal n'est autre que $h_{\text{opt}} = x_0$ (figure ci-contre). S'il y a annulation des dérivées d'ordres élevés impairs sans pour autant avoir affaire à un polynôme, l'identification d'un pas optimal est plus problématique. En revanche, la méthode étant remarquablement précise dans ce cas, l'ajustement du pas aura une influence mineure sur le résultat (à la condition de ne pas utiliser de valeurs fantaisistes bien sûr).

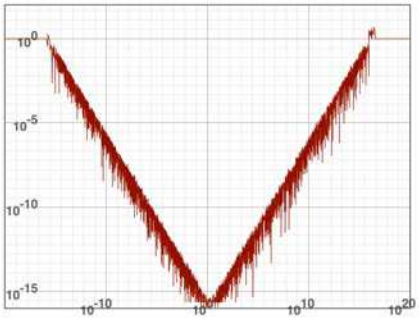


FIGURE 4.2: Erreur sur la différentiation par différences finies d'une fonction P2.

En substituant la valeur usuelle 10^{-16} à ϵ dans 4.4, on trouve finalement :

$$h_{\text{opt}} \simeq c \times 10^{-5} \sqrt[3]{\frac{x_0 |f'(x_0)|}{|f^{(3)}(x_0)|}} \tag{4.5}$$

où la constante $c = \sqrt[3]{0.6} \simeq 0.84$ n'est pas très loin de l'unité.

Revenons à présent à $f = \sin$ et $x_0 = 1$ pour retrouver la valeur $h_{\text{opt}} \simeq 10^{-5}$ déduite graphiquement de 4.1. L'intérêt pratique de 4.5 est bien sûr limité puisque c'est justement pour calculer $f'(x_0)$ que l'on a besoin de h_{opt}. Le but est de montrer par un exemple simple que la valeur générique de [80] est loin d'être optimale, et de suggérer une piste pour en choisir de plus judicieuses. En supposant par exemple le rapport $|f'(x_0)|/\left|f^{(3)}(x_0)\right|$ proche de 1, on détermine une valeur de h_{opt} autour de 10^{-5}. Cette dernière est d'ailleurs plus en adéquation avec ce qui est pratiqué par les numériciens.

D'autre part, même en ayant adopté la valeur optimale du pas, l'erreur sur la dérivée calculée reste sous-optimale. Si on réinjecte 4.5 dans 4.3, on calcule l'erreur minimale réalisable sur une machine. Si on tient compte de l'éventuelle annulation de dérivées d'ordre supérieur, cette erreur prend la forme :

$$e_{\min} = \min \left\{ \quad \epsilon \quad , \quad \left(\frac{|f^{(n)}(x_0)|}{n!} \right)^{1/n} \left(\epsilon x_0 |f'(x_0)|\right)^{\frac{n-1}{n}} \quad \right\} \tag{4.6}$$

En général il y a peu de raisons pour que $f^{(3)}$ s'annule au point considéré, aussi, en prenant $|f'| \simeq 1 \simeq |f^3|$ pour simplifier (mais aussi parce que c'est le cas dans notre exemple 4.1), on trouve $e_{\min} \simeq 10^{-11}$. Ainsi, cinq ordres de magnitude séparent le résultat de la précision machine. Atteindre une telle précision sur la dérivée par la méthode des différences finies nécessite que n soit assez grand pour que $(n!\epsilon)^{-1/n} \leq 1$, c'est-à-dire que toutes les dérivées d'ordre impair jusqu'à l'ordre 17 inclus s'annulent en x_0... Ce n'est pas impossible : les fonctions trigonométriques par exemple annulent toutes leurs dérivées impaires en certains points, mais il ne s'agit que de points isolés. Et pour presque toutes les fonctions (et c'est d'autant plus vrai pour des fonctions issues de modèles mathématiques discrétisés) si cette propriété est vérifiée, ce ne sera qu'en certains points isolés avec lesquels il est bien peu probable d'avoir affaire.

Nous avons donc établi que la plus petite erreur relative que l'on puisse espérer commettre avec les différences finies est d'environ 10^{-11} si on arrive à choisir un bon pas. D'un point de vue numérique il n'y a rien de désastreux. Le problème est, comme on l'a vu, que l'on ne sait déterminer ce pas optimal que dans des cas simples et sans intérêt (puisqu'il faudrait déjà connaître ce que l'on tente d'évaluer). On commettra donc en pratique une erreur en général supérieure. Ce manque de précision est souvent dénoncé par les chantres de la différentiation automatique comme affectant *dramatiquement* la convergence des algorithmes de type quasi-Newton, pouvant parfois entraver le succès de la méthode. Il est pourtant avancé dans [77] que la méthode BFGS par exemple, l'un des plus utilisé de ces algorithmes, présente des propriétés d'autocorrection et qu'à la condition de disposer de la recherche linéaire adéquate, de petites imprécisions sur le gradient ne devraient pas mener à une perte de convergence dans les cas où elle est démontrée. Le problème simple à partir duquel est obtenue la figure 4.3 montre que la réalité se situe entre ces deux positions extrêmes : si l'algorithme converge toujours, même avec des valeurs fantaisistes du pas, ce sont tout de même quatre ordres de magnitude sur la précision de la solution qui sont sacrifiés dans les cas les plus favorables, par rapport à une optimisation avec des dérivées pourvues de la précision machine.

L'autre élément qui peut se révéler en défaveur des différences finies est la complexité algorithmique. Si un calcul séquentiel des quantités 4.1 et 4.2 ne nécessite pas plus d'espace mémoire que celui de f, on voit aisément que la détermination complète des composantes d'un gradient par différences finies aura la complexité de f multipliée par la dimension de

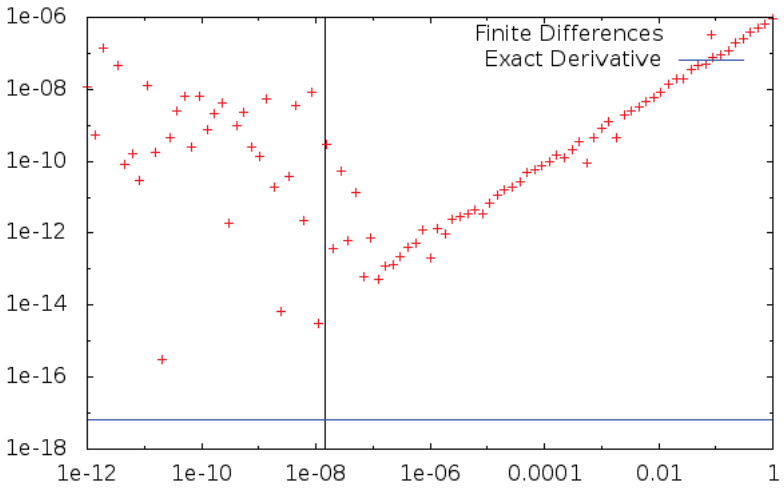


FIGURE 4.3: Distance au minimiseur du dernier itéré dans l'algorithme BFGS en fonction du pas de différences finies pour la minimisation sur $\mathbb{R}^n$ de $x \mapsto \|x\|$, avec $n = 1000$. En bleu, la valeur lorsqu'est fournie une valeur exacte pour la dérivée. Valeur du pas recommandée par[80] : $h = 1,48 \times 10^{-8}$

l'espace des paramètres. Plus précisément, dans le cas des différences *avant*, nous aurons :

$$\mathcal{C}(\nabla f) = (n+1) \times \mathcal{C}(f) \sim n \times \mathcal{C}(f)$$

et pour un calcul par différences centrées :

$$\mathcal{C}(\nabla f) = 2n \times \mathcal{C}(f)$$

Lorsque le coût de l'évaluation de f en terme d'opérations arithmétiques est lourd, ce calcul de dérivées l'est d'autant plus. On a donc là une méthode potentiellement lente et peu précise, donc peu efficace. Nous verrons que la différentiation automatique apporte une amélioration incontestable pour ce qui est de la précision, mais que la question de la réduction du coût informatique est quant à elle plus délicate et, à ce jour, aucune des techniques de différentiation automatique n'apporte de réel progrès en général.

Nous conclurons enfin sur les différences finies en ajoutant qu'elles sont tout de même très utiles, même pour l'amateur de différentiation automatique, ne serait-ce que pour déboguer les calculs de dérivées exploitant d'autres méthodes.

La différentiation symbolique

Cette méthode de différentiation est, dans son essence, très proche de la différentiation automatique. Elle consiste, à peu de choses près, à appliquer les règles usuelles de différentiation à une structure de données représentant avec exactitude l'expression mathématique de la fonction à dériver. Le résultat est une autre instance de cette structure que l'on peut alors évaluer en n'importe quel point pour obtenir les quantités désirées. La structure de données choisie doit être en mesure de décrire exactement l'intégralité de l'algèbre des fonctions susceptibles d'être implémentées par l'utilisateur. Le plus souvent, la génération des dérivées va engendrer des sous-expressions de grande taille qui ne seront pas simplifiables par un procédé complètement automatique. Ces sous-expressions complexes peuvent causer une lourdeur à l'évaluation, ainsi que des erreurs de troncature. Aussi, si ce type de différentiation automatisée est parfois utilisé en calcul scientifique, ce

sera le plus souvent dans le cadre d'une manipulation algébrique des équations du modèle avant implémentation.

Les problèmes soulevés par cette méthode ne se rencontrent pas en *différentiation automatique*, car il n'y a pas de transformation du graphe de la fonction, la dérivée étant calculée en parcourant simplement celui-ci (en sens direct ou inverse selon la méthode employée). L'efficacité du calcul de cette dérivée est donc uniquement liée à l'efficacité de l'implémentation de la fonction initiale, non au hasard des développements, factorisations et simplifications arbitraires d'un automate.

4.1.2 Principe fondamental de la *Différentiation Automatique*

Rappelons le but des techniques de différentiation automatique : partant d'un programme ou d'une partie d'un programme, qui à partir d'un jeu de paramètres d'entrée calcule un ensemble de grandeurs, obtenir un autre programme calculant, si possible, les dérivées des sorties par rapport aux valeurs en entrée. Remarquons que les paramètres dont le type se rapporte directement à un entier ainsi que les chaînes de caractères ne sont évidemment pas de bons candidats par rapport auxquels dériver. Bien souvent, un tel programme sera en pratique un objet relativement complexe, amalgame tortueux d'instructions en tous genres, de sauts conditionnels, de boucles de toutes espèces, d'écrasement et de destruction de variables *etc*... l'inventaire est large, et l'on pourra même parfois rencontrer des opérations non standard complètement hasardeuses, si bien qu'il paraît illusoire d'oser espérer qu'une application ainsi définie puisse être dérivable.

Séquence d'évaluation

Afin de contourner les difficultés soulevées par ces outils et techniques de programmation, nous allons dans un premier temps ne considérer que des programmes dépourvus de ceux-ci. De tels programmes, sont qualifiés d'*abstractions* dans [46] que les auteurs proposent d'appeler *evaluation trace* (en anglais). Cette appellation se justifie par le fait qu'un tel programme, puisqu'il ne doit pas comporter de destruction de variables, conserve en mémoire toutes les variables de toutes les étapes du calcul. C'est en quelque sorte une représentation en séquence d'instructions du graphe de l'expression algébrique de la fonction implémentée. Nous les appellerons quant à nous, des *séquences d'évaluation*.

Mathématiquement, on choisit un ensemble U de fonctions définies sur un sous-domaine de $\mathbb{R}$ dans $\mathbb{R}$ (pour la pratique, les fonctions bénéficiant d'une implémentation dans les bibliothèques standard et qui admettent une dérivée qui s'exprime analytiquement à l'aide de fonctions de U), ainsi qu'un ensemble d'opérateurs binaires B. Un choix minimal serait $B = \{+, \times\}$ si U contient $x \mapsto -x$ et $x \mapsto 1/x$, autrement $B = \{+, -, \times, \text{div }\}$ est le choix le plus naturel. On pourra enfin ajouter $(x, y) \mapsto x^y$ si nécessaire. Nous abordons rapidement les questions soulevées par le cas des fonctions non dérivables dans la section 4.1.3, fonctions dont il peut être difficile de se passer en programmation. Introduisons les quelques notations suivantes, valables pour $1 \leq k \leq N$:

$$\mathcal{C} = \mathbb{R} \times \{\emptyset\} \qquad \mathcal{U}_k = U \times [\![-n, k]\!] \qquad \mathcal{B}_k = B \times [\![-n, k]\!]^2$$

Ces ensembles correspondent respectivement aux choix d'une constante, d'une fonction unaire à laquelle est associé un indice, ou d'une fonction de deux variables avec un couple d'indices. On pose finalement, pour tous $k \in [\![1, N]\!]$:

$$\mathcal{E}_k = \mathcal{C} \cup \mathcal{U}_k \cup \mathcal{B}_k$$

Par analogie avec la programmation, on appellera la donnée d'un entier k et d'un élément (f, I) de $\mathcal{E}_k$ une *instruction*, nos instructions se limitant à la déclaration de variables simultanément à leur définition, par affectation d'une valeur, soit constante indépendante des variables précédemment définies (ensemble $\mathcal{C}$), soit égale à l'image par un élément de U (resp. B) de l'une (resp. de deux) des variables précédentes (l'instruction appartient respectivement à $\mathcal{U}_k$ ou $\mathcal{B}_k$). Nous nommerons par ailleurs *support de l'instruction*, ou seulement *support*, l'objet I, cette fois par analogie au terme employé en analyse fonctionnelle. Le rapprochement entre les deux notions se fera plus évident lors de l'introduction de la différentiation, puisqu'il s'agit ici plus précisément du support de la différentielle de l'instruction en question. Muni de ces écritures, nous proposons alors la définition suivante :

Définition 4.1. *On dit qu'une fonction $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ peut se décomposer en une séquence d'évaluation si il existe une famille $(\varphi_k, I_k)_{1\leq k\leq N}$, avec $\forall k,\ (\varphi_k, I_k) \in \mathcal{E}_{k-1}$, telle que, pour tout $(x_{-n}, \ldots, x_{-1}) \in \mathbb{R}^n$, la suite $(x_k)_{0\leq k\leq N}$, définie par la récurrence finie :*

$$\forall k \in [\![0, N]\!],\ x_k = \widetilde{\varphi_k}\,(x_{-n}, \ldots, x_{k-1})$$

admette pour dernier terme $x_N = f(x_{-n}, \ldots, x_{-1})$. On dira alors que la donnée des $(\varphi_k, I_k)_{0\leq k\leq N}$ est une séquence d'évaluation de f, et on appelera la suite $(x_k)_{k\geq 0}$ la trace d'évaluation de f en $(x_{-n}, \ldots, x_{-1})$ par rapport à $(\varphi_k, I_k)_{0\leq k\leq N}$.

Dans cette définition, $\widetilde{\varphi_k}$ désigne le prolongement constant de φ_k à $\mathbb{R}^{n+k-1}$. Ce prolongement dépend du support I_k et se construit comme suit :

- Si $(\varphi_k, I_k) \in \mathcal{C}$, alors $\varphi_k \in \mathbb{R}$, $\widetilde{\varphi_k}$ se définit simplement comme la fonction constante :

  $$\forall X = (x_{-n}, \ldots, x_{k-1}) \in \mathbb{R}^{n+k-1},\ \widetilde{\varphi_k}(X) = \varphi_k$$

  Ce cas correspond à celui de l'initialisation d'une variable par une valeur constante.
- Si $(\varphi_k, I_k) \in \mathcal{U}_{k-1}$, la fonction est unaire et le support a exactement un élément $I_k = (i)$, on pose alors :

  $$\forall X = (x_{-n}, \ldots, x_{k-1}) \in \mathbb{R}^{n+k-1},\ \widetilde{\varphi_k}(X) = \varphi_k(x_i)$$

- Si $(\varphi_k, I_k) \in \mathcal{B}_{k-1}$, on est confronté à une fonction de deux variables, avec $I_k = (g, d)$, on la prolonge par :

  $$\forall X = (x_{-n}, \ldots, x_{k-1}) \in \mathbb{R}^{n+k-1},\ \widetilde{\varphi_k}(X) = \varphi_k(x_g, x_d)$$

Remarquons tout d'abord que cette définition de la *séquence d'évaluation* s'écarte quelque peu de celle de [46], la notion n'ayant pas fait l'objet d'une utilisation très répandue, nous nous permettons ici de l'adapter aux nécessités de notre exposé. Notre définition semble par ailleurs décrire de façon tortueuse un objet plus aisément transcriptible à l'aide du vocabulaire de la théorie des graphes. Elle a l'avantage d'apporter une vision séquentielle, donc plus proche de la réalité du programmeur. Si la notion de *séquence d'évaluation* présente un intérêt didactique certain, la version du théoricien des graphes aura l'avantage de mettre en avant des structures pouvant se dérouler en parallèle et aura donc plus de succès auprès des amateurs de calcul intensif.

A titre d'exemple, considérons à présent une fonction simple, un peu au hasard :

$$f : (x, y) \in \mathbb{R}^2 \longmapsto \frac{x}{y}\sin(\pi xy) + \ln(x - y - \frac{3}{2})$$

Elle s'écrit aisément en composée de fonctions et d'opérateurs usuels et nous en proposons la séquence d'évaluation 4.1.1.

Séquence d'évaluation 4.1.1 $f(x,y) = \frac{x}{y}\sin(\pi xy) + \ln(x - y - \frac{3}{2})$	
1: $x_{-2} = <$*valeur initiale de* $x>$	*Valeurs test* $x_{-2} = 3$
2: $x_{-1} = <$*valeur initiale de* $y>$	$x_{-1} = 0.5$
3: $x_0 = x_{-2}/x_{-1}$	$x_0 = 6$
4: $x_1 = x_{-2} \times x_{-1}$	$x_1 = 1.5$
5: $x_2 = \pi \times x_1$	$x_2 = 3\pi/2$
6: $x_3 = \sin(x_2)$	$x_2 = -1$
7: $x_4 = x_0 \times x_3$	$x_4 = -6$
8: $x_5 = x_{-2} - x_{-1}$	$x_5 = 2.5$
9: $x_6 = x_5 - 1.5$	$x_6 = 1$
10: $x_7 = \ln(x_6)$	$x_7 = 0$
11: $x_8 = x_4 + x_7$	$x_8 = f(x,y) = -6$

Il n'y a évidemment pas unicité de la trace d'évaluation pour une fonction donnée. On pourra par exemple ajouter un nombre quelconque d'instructions du type $x_k = x_i - x_i,\ i < k$, ou encore de multiplications par 1, multiplications par une variable puis par son inverse, *etc.* Par contre, la longueur N d'une séquence étant bornée inférieurement, elle admet un minimum. On pourra donc définir, pour toute fonction admettant une trace d'évaluation, une classe de ces traces dont les éléments seront optimaux dans le sens où ils minimisent le nombre d'opérations simples nécessaires à l'évaluation de f en un point. Pour se rapprocher encore plus de l'univers de l'informatique, on pourra modifier cette notion d'optimalité de la séquence d'évaluation en pondérant chaque instruction par une grandeur $\mathcal{C}(\varphi_k)$ corrélée au coût informatique de l'opération élémentaire impliquée. Cela permettra de définir un ensemble $\mathcal{S}_f$ de séquences d'évaluation optimales du point de vue du temps de calcul $\sum_{k=0}^{N} \mathcal{C}(\varphi_k)$, auquel nous supposerons que les traces d'évaluation dont il sera question par la suite appartiendront. On pourra alors, lorsqu'il sera nécessaire de faire intervenir des notions en rapport avec la complexité, omettre la dépendance du coût de calcul de f vis-à-vis de la séquence choisie, de sorte que l'on puisse abusivement désigner par $\mathcal{C}(f)$ la quantité $\sum_{k=0}^{N} \mathcal{C}(\varphi_k)$ pour un quelconque élément (φ_k, I_k) de $\mathcal{S}_f$.

La généralisation à des fonctions à valeurs vectorielles est triviale : si F est une fonction de $\mathbb{R}^n$ à valeurs dans $\mathbb{R}^p$ on dira qu'elle admet une trace d'évaluation $(\varphi_k, I_i)_{k \leq N}$ si $\forall x \in \mathbb{R}^n$, les p derniers termes de l'image de x par $(\varphi_k, I_i)_{k \leq N}$ coïncident avec les composantes de $F(x)$. Sauf mention contraire, nous ne considérerons que le cas de fonctions à valeurs réelles afin de ne pas surcharger les notations.

Différentiation directe d'une séquence d'évaluation

Dériver une trace d'évaluation ne présente pas de difficulté particulière. La propriété ci-dessous décrit le procédé de différentiation *direct* des séquences d'évaluation :

Proposition 4.2. *Etant donnée une fonction* $f : \mathbb{R}^n \to \mathbb{R}$ *admettant une séquence d'évaluation* $(\varphi_k, I_k)_{0 \leq k \leq N}$*, on se donne un point* $(x_{-n}, \ldots, x_{-1})$ *de* $\mathbb{R}^n$ *engendrant la trace d'évaluation* $(x_k)_{1 \leq k \leq N}$ *par* $(\varphi_k, I_k)_{0 \leq k \leq N}$ *et on définit la suite finie* $(y_k)_{-n \leq k \leq N}$ *par le*

choix d'un entier $i \in [\![1, n]\!]$ et :

$$\begin{aligned} \forall k \in [\![-n, -1]\!], \quad y_k &= \delta_{-ki} \quad \textit{(initialisation)} \\ \forall k \in [\![0, N]\!], \quad y_k &= \nabla\widetilde{\varphi_k}(x_{-n}, \ldots, x_{k-1}).\begin{pmatrix} y_{-n} \\ \vdots \\ y_{k-1} \end{pmatrix} \\ &= \begin{cases} 0 & si\ I_k = \emptyset \\ \varphi_k'(x_j) & si\ I_k = (j) \\ y_g\partial_1\varphi_k(x_g, x_d) + y_d\partial_2\varphi_k(x_g, x_d) & si\ I_k = (g, d) \end{cases} \end{aligned} \tag{4.7}$$

On a alors :

$$y_N = \frac{\partial f}{\partial x_i}(x_{-n}, \ldots, x_{-1})$$

Démonstration. Elémentaire. On introduit les notations suivantes : pour $k \in [\![0, N]\!]$, on pose $X_k = (x_{-n}, \ldots, x_{k-1}) \in \mathbb{R}^{n+k}$, de même que $Y_k = (y_{-n}, \ldots, y_{k-1})$ et on définit les applications :

$$\begin{array}{rccc} \Phi_k & : & \mathbb{R}^{n+k} & \longrightarrow & \mathbb{R}^{n+k+1} \\ & & x & \longmapsto & (x, \widetilde{\varphi_k}(x)) \end{array}$$

La récurrence de la définition 4.1 s'écrit alors :

$$X_{k+1} = \Phi_k(X_k) \tag{4.8}$$

Et la relation 4.7 de la proposition qui nous intéresse prend la forme :

$$Y_{k+1} = J_{\Phi_k}(X_k)Y_k = \begin{pmatrix} I_{n+k} \\ \hline \nabla\widetilde{\varphi_k}(X_k) \end{pmatrix} Y_k$$

On a donc $Y_{N+1} = J_{\Phi_N} J_{\Phi_{N-1}} \ldots J_{\Phi_0} Y_0 = J_F Y_0$ où $F = \Phi_N \circ \Phi_{N-1} \circ \cdots \circ \Phi_0$. Par construction, la dernière composante de F est égale à la fonction f, on a donc $J_F = \begin{pmatrix} \vdots \\ \hline \nabla f \end{pmatrix}$, ce qui implique que $J_F(X_0)Y_0 = \begin{pmatrix} \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} \vdots \\ \nabla f(X_0).Y_0 \end{pmatrix}$. □

Le code 4.1.2 correspond à l'application de la récurrence 4.7 (proposition 4.2) à la séquence 4.1.1. On remarquera que les quantités calculées pour l'évaluation de f, également nécessaires à celle des dérivées, sont appelées dans le même ordre et au même moment dans les deux programmes. Cela implique que, dans ce cas, on peut se dispenser de la contrainte consistant à ne pas autoriser d'écrasements où de destructions de variables. On pourra par ailleurs facilement appliquer ce mode de différentiation en pratique, puisque les précautions à prendre lors de l'écrasement de variables seront minimales (cas des opérateurs combinant l'affectation à une opération arithmétique comme en `C++`). Nous verrons qu'il est par contre *a priori* vital d'éviter d'écraser les données pour l'autre version de la différentiation automatique ou *mode inverse*.

Séquence d'évaluation 4.1.2 $\frac{\partial f}{\partial x}$ ou $\frac{\partial f}{\partial y}$ par le mode direct de différentiation automatique

1:	$x_2 = <$*valeur initiale de x*$>$	*Valeurs test* $x = 3$
2:	$y_{-2} = <1$ *ou* 0 *selon que l'on calcule respectivement* $\partial_x f$ *ou* $\partial_y f>$	$y_{-2} = 1$ *(resp. 0)*
3:	$x_{-1} = <$*valeur initiale de y*$>$	$y = 0.5$
4:	$y_1 = <0$ *ou 1 pour resp.* $\partial_x f$ *ou* $\partial_y f>$	$y_{-1} = 0$ *(resp. 1)*
5:	$x_0 = x_{-2}/x_{-1}$	$x_0 = 6$
6:	$y_0 = (y_{-2} \times x_{-1} - x_{-2} \times y_{-1})/x_{-1}^2$	$y_0 = 2$ *(resp. -12)*
7:	$x_1 = x_{-2} \times x_{-1}$	$x_1 = 1.5$
8:	$y_1 = y_{-2} \times x_{-1} + x_{-2} \times y_{-1}$	$y_1 = 0.5$ *(resp. 3)*
9:	$x_2 = \pi \times x_1$	$x_2 = 3\pi/2$
10:	$y_2 = \pi \times y_1$	$y_2 = \pi/2$ *(resp.* 3π*)*
11:	$x_3 = \sin(x_2)$	$x_2 = -1$
12:	$y_3 = y_2 \cos(x_2)$	$y_3 = 0$ *(resp. 0)*
13:	$x_4 = x_0 \times x_3$	$x_4 = -6$
14:	$y_4 = y_0 \times x_3 + x_0 \times y_3$	$y_4 = -6$ *(resp. 12)*
15:	$x_5 = x_{-2} - x_{-1}$	$x_5 = 2.5$
16:	$y_5 = y_{-2} - y_{-1}$	$y_5 = 1$ *(resp. -1)*
17:	$x_6 = x_5 - 1.5$	$x_6 = 1$
18:	$y_6 = y_5$	$y_6 = 1$ *(resp. -1)*
19:	$x_7 = \ln(x_6)$	$x_7 = 0$
20:	$y_7 = y_7/x_7$	$y_7 = 1$ *(resp. -1)*
21:	$x_8 = x_4 + x_7$	$x_8 = f(x, y) = -6$
22:	$y_8 = y_4 + y_7$	$y_8 = \frac{\partial f}{\partial x}(x, y) = -1$ *(resp. 11)*

On conçoit aisément comment généraliser cette méthode pour évaluer non pas une, mais toutes les dérivées partielles de f en un point donné : on remplacerait dans la proposition 4.2 les réels y_k par des vecteurs lignes Y_k de $\mathbb{R}^n$ et :

$$\begin{aligned} \forall k \in [\![-n, -1]\!], \quad & Y_k = (\delta_{-ki})_{1 \leq i \leq n} = e_{-k} \\ \forall k \in [\![0, N]\!], \quad & Y_k = \nabla \widetilde{\varphi_k}(x_{-n}, \ldots, x_{k-1}) \underbrace{\begin{pmatrix} Y_{-n} \\ \vdots \\ Y_{k-1} \end{pmatrix}}_{\in \mathcal{M}_{n+k,n}(\mathbb{R})} \end{aligned} \tag{4.9}$$

En pratique, pour des raisons évidentes de performance, le produit matrice-vecteur ci-dessus sera réécrit sous la forme suivante :

$$Y_k = \begin{cases} 0 & \text{si } I_k = \emptyset \\ \varphi_k'(x_i) Y_i & \text{si } I_k = (i) \\ \partial_1 \varphi_k(x_g, x_d) Y_g + \partial_2 \varphi_k(x_g, x_d) Y_d & \text{si } I_k = (g, d) \end{cases}$$

Là encore, on peut facilement vérifier que le dernier terme de la suite définie dans l'équation 4.9 ci-dessus est tel que :

$$Y_N = \nabla f(x_{-n}, \ldots, x_{-1})$$

Et dans le cas où f est à valeurs dans $\mathbb{R}^p$, les vecteurs $Y_{N-k},\ 0 \leq k \leq p-1$ sont les lignes

de la jacobienne de f en $(x_{-n}, \ldots, x_{-1})$:

$$\begin{pmatrix} Y_{N-p+1} \\ \vdots \\ Y_N \end{pmatrix} = J_f(x_{-n}, \ldots, x_{-1})$$

Différentiation en mode inverse/adjoint

Cette méthode trouve ses fondements dans le fait que, étant donnée une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$, l'opérateur adjoint de la différentielle de f en un point $x \in \mathbb{R}^n$ définit une application linéaire de $\mathbb{R}^m$ dans $\mathbb{R}^n$ (ces espaces s'identifiant à leur dual respectif) : $y \mapsto df(x)^* y$. La matrice associée à $df(x)^*$ est la transposée de la jacobienne évaluée en x, on pourra donc écrire :

$$\begin{array}{rccc} df(x)^* \ : & \mathbb{R}^m & \longrightarrow & \mathbb{R}^n \\ & y & \longmapsto & J_f(x)^{\mathrm{T}}\, y \end{array}$$

Partant de ce constat, la plupart des ouvrages concernant ce sujet en arrivent plus ou moins directement à un algorithme de dérivation de code. Dans tous les cas, les justifications nous ont semblé peu satisfaisantes et hermétiques ; nous proposons donc le développement mathématique suivant qui, bien qu'un peu lourd, ne fait appel qu'à des notions élémentaires et n'a donc de compliqué que l'aspect. Ce n'est qu'après avoir établi ces formules que nous avons pu aboutir à une réelle compréhension de ce mode de différentiation automatique et, toutes les références sur le sujet nous ayant laissé perplexe, ce petit travail nous paraît jeter un peu de clarté sur un sujet plutôt obscur.

On se restreint à nouveau au cas où f est à valeurs dans $\mathbb{R}$ et admet une trace d'évaluation $(\varphi_k, I_k)_{0 \le k \le N}$. Et comme dans la définition 4.1, on se donne un point $(x_{-n}, \ldots, x_{-1})$ de $\mathbb{R}^n$ et on note $(x_k)_{k \ge 0}$ son image par $(\varphi_k, I_k)_{0 \le k \le N}$.

On reprend les notations introduites pour la preuve de la proposition 4.2 en modifiant quelque peu celle des vecteurs $Y_k \in \mathbb{R}^{n+k+1}$ par le procédé récursif fini descendant suivant :

$$\begin{array}{c} Y_N = (0, \ldots, 0, 1) \in \mathbb{R}^{n+N+1} \\ \forall k \in [\![0, N]\!] \ , \ Y_{k-1} = J_{\Phi_k}(X_k)^{\mathrm{T}}\, Y_k \end{array} \tag{4.10}$$

De par la définition des fonctions Φ_k, on a en outre :

$$J_{\Phi_k}{}^{\mathrm{T}} = \left(\ I_{n+k} \ \middle| \ \nabla \widetilde{\varphi_k}^{\mathrm{T}} \ \right)$$

On aura donc, en posant $Y_k = (y_k^j)_{-n \le j \le k}$:

$$\begin{array}{rcl} Y_{k-1} & = & \left(\ I_{n+k} \ \middle| \ \nabla \widetilde{\varphi_k}(X_k)^{\mathrm{T}} \ \right) Y_k \\ & = & \left(y_k^j + y_k^k \partial_j \widetilde{\varphi_k}(X_k) \right)_{-n \le j \le k-1} \end{array}$$

On voit donc que pour $j \in [\![-n, N]\!]$ fixé, les termes y_k^j sont les sommes partielles de $y_j^j = \sum_{k>j} y_k^k \partial_j \widetilde{\varphi_k}(X_k)$. D'un point de vue algorithmique, on n'aura donc à stocker qu'une unique variable pour chaque suite finie $(y_k^j)_{j \le k \le N}$ qui sera mise à jour par incrémentations successives à mesure que l'on parcourt les dérivées des φ_k dans le sens inverse de celui de la trace d'évaluation initiale. On a par ailleurs le résultat suivant :

Proposition 4.3. $\forall i \ 1 \le i \le n, \ y_{-1}^{-i} = \dfrac{\partial f}{\partial x_i}(x_{-n}, \ldots, x_{-1})$

Démonstration. Quasiment identique à celle de la proposition 4.2 : par construction, la $(n+N)$-ème composante de $F = \Phi_N \circ \Phi_{N-1} \circ \cdots \circ \Phi_0$ est égale à f, on a donc pour tout $x \in \mathbb{R}^n$, $J_F(x)^{\mathrm{T}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} = \nabla f(x)^{\mathrm{T}}$. Et comme $J_F(x)^{\mathrm{T}} = J_{\Phi_0}(x)^{\mathrm{T}}\, J_{\Phi_1}(x)^{\mathrm{T}} \ldots J_{\Phi_N}(x)^{\mathrm{T}}$, on a $Y_{-1} = \nabla f(x)^{\mathrm{T}}$ □

Les quantités pertinentes dans le procédé de dérivation développé ci-dessus sont donc les valeurs $\lambda_j = y_j^j$ si $j \geq 0$ et $\lambda_j = y_{-1}^j$ pour $j \in [\![-n, -1]\!]$. La différentiation automatique en *mode adjoint* (ou *mode inverse*) de la trace d'évaluation $(\varphi_k, I_k)_{0 \leq k \leq N}$ au point $(x_{-n}, \ldots, x_{-1})$ consiste alors à construire la suite finie $(\lambda_j)_{-n \leq j \leq N}$ par le processus récursif descendant suivant :

$$\lambda_N = 1 \text{ et } \forall j \in [\![0, N-1]\!],\ \lambda_j = \sum_{k>j} \lambda_k \partial_j \widetilde{\varphi_k}(X_k) \tag{4.11}$$

De cette proposition 4.3, on déduit immédiatement que les dérivées partielles de f évaluées en $(x_{-n}, \ldots, x_{-1})$ sont *accumulées* dans $\lambda_{-n}, \ldots, \lambda_{-1}$. On relèvera donc un net avantage de ce mode de différentiation par rapport au mode *direct* : ce dernier nécessite d'adjoindre à chaque variable de la trace d'évaluation autant de variables qu'il y a de directions de dérivation, alors que le mode *inverse* n'a besoin *a priori* que d'une variable supplémentaire quel que soit le nombre de directions. Appliquons les principes énoncés ci-dessus à l'exemple 4.1.1. La notation "$a \mathrel{+}= b$" signifie $a := a + b$ comme pour les opérateurs cumulant affectation et opération en `C++` .

Séquence d'évaluation 4.2.3 $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial y}$ par le mode inverse/adjoint de différentiation automatique

---

1:	$x_{-2} =$ <*valeur initiale de x*>	*Valeurs test* $x_{-2} = 3$
2:	$x_{-1} =$ <*valeur initiale de y*>	$x_{-1} = 0.5$
3:	$x_0 = x/y$	$x_0 = 6$
4:	$x_1 = x_{-2} \times x_{-1}$	$x_1 = 1.5$
5:	$x_2 = \pi \times x_1$	$x_2 = 3\pi/2$
6:	$x_3 = \sin(x_2)$	$x_2 = -1$
7:	$x_4 = x_0 \times x_3$	$x_4 = -6$
8:	$x_5 = x_{-2} - x_{-1}$	$x_5 = 2.5$
9:	$x_6 = x_5 - 1.5$	$x_6 = 1$
10:	$x_7 = \ln(x_6)$	$x_7 = 0$
11:	$x_8 = x_4 + x_7$	$x_8 = f(x, y) = -6$
12:	$\lambda_8 = 1$	*Sélection de la quantité à différentier*
13:	**for** $i = -2$ **to** 7 **do**	
14:	$\lambda_i = 0$	*Initialisation des valeurs adjointes*
15:	**end for**	
16:	$\lambda_7 \mathrel{+}= \lambda_8 \times 1$	*l. 11*

17:	$\lambda_4 \mathrel{+}= \lambda_8 \times 1$	
18:	$\lambda_6 \mathrel{+}= \lambda_7 \times 1/x_6$	*l. 10*
19:	$\lambda_5 \mathrel{+}= \lambda_6 \times 1$	*l. 9*
20:	$\lambda_{-1} \mathrel{+}= \lambda_5 \times 1$	*l. 8*
21:	$\lambda_{-2} \mathrel{+}= \lambda_5 \times (-1)$	
22:	$\lambda_3 \mathrel{+}= \lambda_4 \times x_0$	*l. 7*
23:	$\lambda_0 \mathrel{+}= \lambda_4 \times x_3$	
24:	$\lambda_2 \mathrel{+}= \lambda_3 \times \cos(x_2)$	*l. 6*
25:	$\lambda_1 \mathrel{+}= \lambda_2 \times \pi$	*l. 5*
26:	$\lambda_{-1} \mathrel{+}= \lambda_1 \times x_{-2}$	*l. 4*
27:	$\lambda_2 \mathrel{+}= \lambda_1 \times x_{-1}$	
28:	$\lambda_{-1} \mathrel{+}= \lambda_0 \times (-x_{-2})/x_{-1}^2$	*l. 3*
29:	$\lambda_{-2} \mathrel{+}= \lambda_0 \times 1/x_{-1}$	$\lambda_{-2} = -1,\ \lambda_{-1} = 11$

Un lecteur averti aura remarqué que pour calculer les λ_j, on aura besoin des x_j dans l'ordre inverse où ces derniers sont calculés par le programme initial. Dans le cas de nos *traces d'évaluation* cela n'apporte pas de complication particulière puisque dans une implémentation d'une telle fonction, les variables ne sont jamais écrasées et sont toujours accessibles. C'est en revanche bien plus problématique dans la réalité où l'on ne rencontre jamais de programme possédant de telles caractéristiques. Trois alternatives se présentent alors :

- conserver une copie de la valeur de chaque variable à chaque fois qu'elle est écrasée et lorsqu'elle est détruite
- ré-exécuter le programme pour recalculer les valeurs écrasées et les variables détruites
- un savant mélange des deux propositions précédentes

Nous verrons dans la section 4.1.5 quel est le coût selon la stratégie adoptée pour obtenir à nouveau ou conserver ces valeurs.

La généralisation de cette méthode à une fonction à valeur dans $\mathbb{R}^m$ se fait en remplaçant les λ_i dans 4.11 par des vecteurs de $\mathbb{R}^m$, initialisés par les vecteurs $(e_j)_{1\leq j\leq m}$ de la base canonique de $\mathbb{R}^m$:

$$\forall 1 \leq j \leq m,\ \Lambda_{N-m+j} = e_j \ \text{ , et } \forall j \in [\![0, N-1]\!],\ \Lambda_j = \sum_{k>j} \Lambda_k \partial_j \widetilde{\varphi_k}(X_k) \tag{4.12}$$

Le calcul aboutit sur les colonnes de la jacobienne de f : $\begin{pmatrix}\Lambda_{-n} & \dots & \Lambda_{-1}\end{pmatrix} = J_f(x_{-n}, \dots, x_{-1})$.

Le programme vu comme une fonction implicite

Il est également intéressant de considérer les données calculées par un programme comme la (ou les) solution(s) d'un gros système d'équations. C'est là une approche courante dans la littérature et on en trouvera un petit exposé assez clair dans [91], ainsi qu'une présentation détaillée puis largement exploitée dans [46]. Comme cette représentation permet de faire le lien avec certaines opportunités d'optimisation de la complexité algorithmique des méthodes de différentiation automatique, nous allons la préciser quelque peu dans le cadre de nos séquences d'évaluation.

Si on dispose pour la fonction implémentée $f : \mathbb{R}^n \to \mathbb{R}^m$ d'une séquence d'évaluation $(\varphi_k, I_k)_{0\leq k\leq N}$, alors la trace d'évaluation associée aux variables d'entrée $x = (x_{-n}, \dots, x_{-1})$ est la solution en les $(y_i)_{-n\leq i\leq N}$ du vaste système non linéaire ci-dessous :

$$E(x, y_{-n}, \dots, y_N) = 0 \tag{4.13}$$

où $E = (E_k)_{-n \leq j \leq N} : \mathbb{R}^n \times \mathbb{R}^{N+n} \longrightarrow \mathbb{R}^{N+n}$ est définie par la donnée des $N+n$ composantes :

$$E_k : (x, y_{-n}, \ldots, y_N) \longmapsto \begin{cases} x_k - y_k & \text{si } k \in [\![-n, -1]\!] \\ \widetilde{\varphi_k}(y_{-n}, \ldots, y_{k-1}) - y_k & \text{si } k \in [\![0, N]\!] \end{cases}$$

On supposera par ailleurs que les m dernières composantes ne dépendent que de la variable de même indice et de variables d'indices strictement inférieurs à $N - m$. Cela implique que les m derniers éléments de la séquence d'évaluation $(x_{N-m+1}, \ldots, x_N) = f(x)$ et les sorties du programme sont indépendants les uns des autres. Ce n'est pas très contraignant, et si nécessaire il faudra seulement ajouter quelques affectations sans autre opération à la séquence d'évaluation. On en déduit que la matrice jacobienne obtenue par dérivation de E par rapport aux variables $y_{-n}, \ldots, y_N$ est triangulaire inférieure avec la représentation par blocs suivante :

$$\frac{\partial E}{\partial y} = \begin{pmatrix} -I_n & 0 & 0 \\ B & L - I_{N-m} & 0 \\ R & T & -I_m \end{pmatrix}$$

où l'on aura omis la dépendance vis-à-vis de x et $(y_{-n}, \ldots, y_N)$ de la matrice strictement triangulaire inférieure L, et des matrices B, R et T. On en déduit au passage que le déterminant de $\dfrac{\partial E}{\partial y}$ est toujours non nul, ce qui permet de vérifier par le théorème des fonctions implicites que le système 4.13 définit bien une fonction de $\mathbb{R}^n$ dans $\mathbb{R}^{N+n}$, et on obtient par différentiation implicite :

$$\frac{\partial y}{\partial x} = \frac{\partial E}{\partial y}^{-1} \times \frac{\partial E}{\partial x} = \underbrace{\begin{pmatrix} -I_n & 0 & 0 \\ B & L - I_{N-m} & 0 \\ R & T & -I_m \end{pmatrix}^{-1}}_{\in \mathcal{M}_{n+N}(\mathbb{R})} \times \underbrace{\begin{pmatrix} I_n \\ 0 \\ 0 \end{pmatrix}}_{\in \mathcal{M}_{n+N,n}(\mathbb{R})}$$

De cette égalité découle l'expression suivante des dérivées des m dernières variables y_i par rapport aux entrées x :

$$J_f(x) = R + T(I_{N-m} - L)^{-1} B \tag{4.14}$$

Comme remarqué dans [91], le mode direct consiste à trouver une matrice $W \in \mathcal{M}_{N-m,n}(\mathbb{R})$ en résolvant par substitution la suite de systèmes triangulaires inférieurs engendrée par $(I_{N-m} - L)W = B$, puis à calculer $J_f = R + TW$. Le mode inverse revient quant à lui à trouver $Z \in \mathcal{M}_{n,N-m}(\mathbb{R})$ telle que $(I - L)^{\mathrm{T}} Z = T^{\mathrm{T}}$ avant de calculer $J_f = R + Z^{\mathrm{T}} B$. Dans ces deux cas, les matrices dont il est question ne sont jamais assemblées, puisque bien qu'étant très creuses (pas plus de deux éléments non nuls par ligne), on a affaire à des dimensions dont les ordres de grandeurs avoisinent ceux de l'astronomie. En effet, pour un programme dont le fonctionnement s'étale sur une dizaine de minutes, avec une puissance de calcul en centaines de millions d'opérations par seconde, simulation relativement modeste donc, la valeur correspondante de N est de l'ordre de la puissance onzième de 10. On comprend donc pourquoi personne n'osera en pratique utiliser ces matrices. Remarquons à ce propos que différentier un programme en mode inverse sans utiliser de balisage (voir plus loin) revient en quelque sorte à construire explicitement ces objets colossaux. Cette écriture sous forme de système non linéaire de la séquence d'évaluation est plutôt utilisée dans la perspective d'établir certains résultats d'ordre théorique.

4.1.3 Fonctions non dérivables

Nous avons jusqu'ici supposé que les fonctions admissibles pour définir une *séquence d'évaluation* sont au moins deux fois dérivables, et ainsi implicitement considéré que la *différentiation automatique* ne peut s'appliquer qu'à l'implémentation de fonctions admettant une *séquence d'évaluation* construite à partir de fonctions C^2. Il s'agit cependant d'une hypothèse peu réaliste car, au-delà de l'appel explicite à des routines implémentant des fonctions évidemment non dérivables en certains points telles que la valeur absolue, le minimum ou le maximum, ou encore la racine carrée, l'immense majorité des programmes comporte des instructions conditionnelles susceptibles d'introduire implicitement de la non-dérivabilité, voire des discontinuités dans le programme.

Dans le pseudo-code qui suit, nous présentons quelques instructions dont l'introduction dans un programme mènera à des problèmes de dérivabilité. Les variables w, x et y étant celles par rapport auxquelles on pourrait souhaiter différentier les autres variables :

1: $z_1 = |x|\,;\ z_2 = \min(x, y)\,;\ z_3 = \max(x, y)$
2: $z_3 = \sqrt{x}\,;$ *(si évaluée pour $x = 0$)*
3: $z_4 = \arcsin(x)\,;$ *(ou* arccos*, en cas d'évaluation en -1 ou 1, et* arctan *en 0)*
4: $z_5 = w > 0\ ?\ x\ :\ y\,;$ *(l'opérateur conditionnel du C++ ,...*
5:
6: **if** $w > 0$ **then**
7: $\quad z_6 = x$ *... équivalent à ces cinq lignes)*
8: **else**
9: $\quad z_6 = y$
10: **end if**

Remarque 4.4. En dehors des fonctions telles que $\sqrt{}$, arcsin, arccos, *etc...*, toute fonction impliquant des comparaisons peut être exprimée à l'aide des seules valeur absolue et fonction de Heavyside $H : x(\mapsto x < 0\ ?\ 0\ :\ 1)$. On a en effet : $\max(x, y) = \frac{1}{2}(x + y + |x - y|)$, $\min(x, y) = \frac{1}{2}(x + y - |x - y|)$ et $z = w > 0\ ?\ x\ :\ y \Leftrightarrow z = xH(w) + y(1 - H(w))$. On pourra donc se restreindre à l'étude du traitement de ces deux fonctions en termes de différentiation automatique (plus celui des fonctions mathématiques présentant des dérivées singulières).

Dans tous les cas, les fonctions élémentaires pouvant poser des problèmes de dérivation ne présentent qu'un unique point en lequel elles sont discontinues ou non différentiables. Toute fonction que l'on peut implémenter à l'aide de ce genre de fonction sera donc dérivable *presque partout*, voire C^∞ presque partout. On pourra donc, dans un premier temps simplement ignorer ces points sensibles puisque la probabilité d'évaluer en ceux-ci est *a priori* nulle. Cette approche est néanmoins maladroite et, comme le remarque [46], un outil de différentiation automatique qui ne fournirait pas un minimum d'information quant à la fiabilité des dérivées qu'il calcule serait tout aussi dangereux à utiliser qu'un solveur linéaire qui n'avertirait pas l'utilisateur lorsque celui-ci tente de résoudre un système presque singulier ou mal conditionné. Une solution alliant simplicité et un minimum de sécurité consiste à simplement tester, à chaque appel de fonctions singulières du point de vue de la différentiation, si l'évaluation se fait suffisamment loin des points à problème. On pourra éventuellement raffiner cette méthode en calculant un indicateur de fiabilité (voir la section 4.2).

Le traitement de fonctions lipschitziennes, par l'introduction de la notion de différentielles généralisées (voir [30]) n'a à ce jour jamais été implémenté, principalement du fait de la complexité des objets proposés pour la généralisation des dérivées de ce type de fonction. Rappelons à ce propos la généralisation proposée dans l'ouvrage précédemment

cité, où le gradient généralisé en un point $x \in \Omega \subset \mathbb{R}^n$ d'une fonction lipschitzienne f définie sur Ω est : f étant dérivable presque partout d'après le théorème de Rademacher, on note $S \subset \Omega$ l'ensemble des points en lesquels f n'admet pas de dérivée au sens usuel et :

$$\partial f(x) = \text{Conv}\left\{ g \in \mathbb{R}^n \;\middle|\; \exists (x_k)_{k\in\mathbb{N}} \in (\Omega \backslash S)^{\mathbb{N}} \text{ telle que } \lim_{k\to+\infty} x_k = x \text{ et } g = \lim_{k\to+\infty} \nabla f(x_k) \right\}$$

Il n'existe pas à ce jour de travaux ayant permis de dégager des structures de données permettant la description exacte des objets mathématiques impliqués et qui n'induiraient pas un traitement additionnel coûteux. De plus, les règles de dérivation des opérations usuelles dans les théories proposées ne se généralisent que sous forme d'inclusions. Il y aurait donc perte d'information à mesure que sont composées fonctions et opérations algébriques, si bien que le résultat obtenu par application de ces règles de dérivation à toute une *séquence d'évaluation*, même de taille modeste, serait difficilement exploitable. L'exemple de la fonction $f : x \mapsto |x| - |x| = 0$, trouvé dans [46], est très éloquent : par dérivation de la fonction nulle en 0 on a $\partial f(0) = \{0\}$, cependant l'application de la règle de dérivation des combinaisons linéaires de fonctions ([30]) se réduit à $\partial f(0) \subset [-2, 2]$. Il est donc peu probable que ces objets mathématiques trouvent leur place en différentiation automatique, d'autant plus que leur utilité en simulation numérique n'est pas affirmée. Pour ce qui est du traitement de la non-différentiabilité, les outils de différentiation automatique les plus performants se limitent aux calculs de dérivées directionnelles lorsqu'il y a évaluation en des points singuliers, avec, éventuellement, gestion des infinis pour les outils les plus robustes.

4.1.4 Dérivées d'ordres supérieurs

Le calcul de dérivées à des ordres élevés (en matière de dérivation on entend par ordre élevé tout ordre au moins égal à deux) peut se révéler problématique, compte tenu de la complexité potentiellement explosive des méthodes de différentiation automatique (voir section 4.1.5). Les objets mathématiques en rapport avec la différentiation à hauts ordres les plus simples à obtenir sont les coefficients des séries de Taylor tronquées comme nous allons le voir. La question du calcul des coefficients de tenseurs différentiels est plus délicate et nous nous bornerons à une rapide analyse du cas de la matrice hessienne.

Polynômes de Taylor en différentiation automatique

Donnons nous à nouveau une fonction f et une séquence d'évaluation $(\varphi_k, I_k)_{0\leq k\leq N}$. Etant donnés un jeu de paramètres d'entrée $X = (x_{-n}, \ldots, x_{-1})$ et une direction $u \in \mathbb{R}^n$, il s'agit de déterminer les coefficients de la série de Taylor de f dans la direction u, tronquée à un ordre donné $d \in \mathbb{N}^*$:

$$\forall h \in \mathbb{R},\ f(X + hu) = \sum_{i=0}^{d} f_i h^i \tag{4.15}$$

On se limite à une différentiation automatique en mode *direct*, aussi doit-on déterminer comment enrichir chaque instruction de la séquence d'évaluation pour arriver à ce but.

Pour cela, à chaque variable x_k va être associée une famille finie $(y_k^i)_{0\leq i\leq d}$ (avec $y_k^0 = x_k$) correspondant aux coefficients du développement en série de Taylor en X de la fonction implémentée par la séquence d'évaluation partielle $(\varphi_j, I_j)_{0\leq j\leq k}$. Par abus d'écriture, nous confondrons cette fonction avec l'élément correspondant dans la trace d'évaluation, et nous

irons même plus loin dans les simplifications en écrivant $x_k(h)$ pour en réalité désigner $x_k(X + hu)$. Les variables d'entrée $(x_{-n}, \ldots, x_{-1})$ se voient augmentées de coefficients de Taylor tous nuls, à l'exception des deux premiers :

$$\forall k \in [\![1, n]\!],\ y^0_{-k} = x_{-k},\ y^1_{-k} = u_k \text{ et } y^i_{-k} = 0\ (i \geq 2) \tag{4.16}$$

Ceci révèle que la variable d'entrée n'est plus X mais $X+hu$, qui est affine en h et n'admet donc des coefficients non nuls qu'aux ordres 0 et 1 dans son développement en série de Taylor.

Les coefficients de Taylor associés à x_k sont alors calculés à partir de ceux de x_{k-1}, comme l'étaient les dérivées partielles dans le mode direct, à l'exception du cas où φ_k est une constante, pour lequel les y^i_k sont uniformément annulés pour tout i. C'est très simple si φ_k est un opérateur binaire, avec $I_k = (l, r)$ (pour *left* et *right*), les mises à jour se font comme suit :

Opérateur	Relation ($\forall i \in [\![0, d]\!]$)
$\pm$	$y^i_k = y^i_l \pm y^i_r$
$\times$	$y^i_k = \sum_{j=0}^{i} y^j_l y^{i-j}_r$
div	$y^i_k = \frac{1}{x_r}\left(y^i_l - \sum_{j=0}^{i-1} y^j_k y^{i-j}_r\right)$

(4.17)

Le calcul des termes associés à une multiplication est donc un simple produit de convolution discret. Addition et soustraction restent des opérations triviales puisqu'elles conduisent à appliquer respectivement la même opération terme à terme. Ces cas ne posent donc aucun problème et peuvent de surcroît faire facilement l'objet d'une implémentation en parallèle. La relation obtenue pour l'opérateur de division est quant à elle plus retorse car il s'agit d'une véritable relation de récurrence. C'est également le cas pour la fonction racine carrée, dont la relation est déterminée par l'observation que $u = \sqrt{v} \Rightarrow u^2 = v$

Fonction	Relation ($\forall i \in [\![0, d]\!]$)
$x_k = \sqrt{x_p}$	$y^i_k = \frac{1}{2x_p}\left(y^i_p - \sum_{j=1}^{i-1} y^j_k y^{i-j}_k\right)$

La mise à jour des coefficients obtenus après application des autres fonctions à une unique variable est plus délicate. Si par exemple, l'instruction k est $x_k = \varphi_k(x_p)$ $(I_k = (p))$, la tentation serait grande d'écrire le développement en série de Taylor de φ_k puis d'y introduire $x_p(h)$. On se rendrait bien vite compte que les expressions obtenues seraient fort complexes, avec des opérations instables d'un point de vue numérique.

Il est donc nécessaire de trouver un procédé plus astucieux. Penchons-nous à titre d'exemple sur le cas de la fonction exponentielle, en supposant que $(\varphi_k, I_k) = (exp, p)$. Si l'on pense la variable x_k comme une fonction de la seule variable h, et que l'on dérive l'instruction k, on obtient :

$$x'_k(h) = x'_p(h) \exp(x_p(h)) = x'_p(h) x_k(h)$$

Le décalage sur l'ordre des coefficients introduit par la dérivée va permettre d'établir une relation de récurrence. Posons $\tilde{y}^i_k = i y^i_k$ (idem pour y^i_p), on aura alors, compte tenu de l'équation ci-dessus :

$$\forall i \geq 1,\ \tilde{y}^i_k = \sum_{j=1}^{i} \tilde{y}^j_p y^{i-j}_k \tag{4.18}$$

Toutes les fonctions ne présentant pas une propriété aussi remarquable que la fonction exponentielle en termes de dérivation, examinons le cas de celles dont la dérivée s'exprime uniquement en fonction de la variable et des opérations usuelles, et éventuellement de fonctions dont on connaît le développement (inverse, racine carrée, puissances entières). Si, cette fois, $(\varphi_k, I_k) = (\ln, p)$:

$$x'_k(h) = \frac{x'_p(h)}{x_p(h)}$$

Ce qui donne, d'après 4.17 :

$$\forall i \geq 2,\ \tilde{y}^i_k = \frac{1}{x_p}\left(\tilde{y}^i_p - \sum_{j=1}^{i-1} y_p^{i-j}\tilde{y}^j_p\right)$$

Complétons enfin cette étude par le cas des fonctions trigonométriques, qui nécessitent un traitement de faveur puisqu'elles ne vérifient pas d'équation différentielle d'ordre 1 à coefficients simples. Comme $\sin' = \cos$ et $\cos' = -\sin$, dès que $\varphi_k \in \{\cos, \sin\}$, on calculera les deux familles de coefficients $(\gamma^i)_{1\leq i\leq d}$ et $(\sigma^i)_{1\leq i\leq d}$ respectivement associées à $c_k = \cos(x_p)$ et $s_k = \sin(x_p)$, par la double récurrence qui suit :

$$\begin{aligned}\tilde{\sigma}^i &= \sum_{j=1}^{i} \tilde{y}^j_p \gamma^{i-j} \\ \tilde{\gamma}^i &= \sum_{j=1}^{i} -\tilde{y}^j_p \sigma^{i-j}\end{aligned}$$

On trouvera dans [46] de nombreux éléments de réflexion au sujet des questions de performance.

Tenseurs différentiels d'ordres élevés et hessienne

Au lieu de privilégier une direction, on pourrait s'intéresser au cas d'un petit sous-espace de $\mathbb{R}^n$ engendré par une petite famille de p vecteurs dont la collection formerait une matrice $D \in \mathcal{M}_{n,p}(\mathbb{R})$. Etant donné un point $X \in \mathbb{R}^n$, on considérerait alors l'application $h \in \mathbb{R}^p \longmapsto f(X + Dh)$ afin de généraliser le procédé précédemment exposé, pour obtenir des relations de récurrence sur les coefficients des tenseurs différentiels de cette restriction jusqu'à un ordre arbitraire d. On choisira en général p raisonnablement petit, et sûrement pas proche de n : le nombre de variables de la fonction initiale f pouvant être grand, il est vain d'espérer pouvoir effectuer un calcul complet des tenseurs différentiels associés à celle-ci à des ordres élevés (*i.e.* supérieurs à 2). Nous ne développerons pas ici cette méthode, renvoyant le lecteur aux ouvrages généraux sur la différentiation automatique.

Précisons toutefois que pour le calcul de la matrice hessienne dans son intégralité par application successive des deux modes de différentiation automatique, l'usage est d'appliquer le mode direct à un programme ayant été préalablement différencié par un procédé de type inverse. On a vu que le mode inverse se montrait plus efficace lorsque la fonction différentiée est à valeurs dans un espace de petite dimension sur $\mathbb{R}$ (idéalement à valeurs dans $\mathbb{R}$). Il est donc beaucoup moins astucieux d'effectuer une différentiation en mode inverse sur un programme calculant un gradient obtenu par le mode direct. Pour ce qui est de la composition successive du même mode de différentiation, elle peut fonctionner avec le mode direct quand n est très petit et est inconcevable avec le mode inverse pour des raisons évidentes de complexité (aussi bien en mémoire qu'en coût de calcul).

4.1.5 Complexité en temps et en espace des deux modes de différentiation automatique

Comme toujours lorsque l'outil informatique est impliqué, il est important de connaître la complexité en temps et en mémoire des méthodes déployées. L'analyse des méthodes de différentiation automatique est relativement simple dans le cas du mode direct, et à peine plus complexe pour ce qui est du mode inverse. Il s'agit bien entendu d'analyses de complexité pour les versions naïves des modes de différentiation, c'est-à-dire tels qu'ils apparaissent dans la section 4.1.2. Les recherches actuelles dans le domaine de la différentiation automatique visent à réduire ces coûts qui, comparés à ceux engendrés par un code implémentant une dérivée calculée *à la main*, manifestent des qualités très modestes pour ce qui est des performances, comme nous allons le voir.

Afin de procéder à cette étude de complexité, nous distinguerons les opérations $+,\times$ et les appels à des routines implémentant des fonctions non linéaires. Nous noterons respectivement $\mathrm{ADD}(f)$, $\mathrm{MUL}(f)$ et $\mathrm{ONL}(f)$ les nombres de ces opérations que nécessite une implémentation fixée d'une fonction f. On doit donc avoir $\mathcal{C}(f)$ proportionnel à une combinaison linéaire de ces quatre dernières grandeurs. On reprendra enfin les notations introduites dans la section 4.1.2.

Complexité du mode direct

Dans sa version la plus naïve, toute variable de l'implémentation de f est augmentée d'un vecteur à n composantes. Cette méthode ne demandant par ailleurs pas de modification dans la portée des variables, il n'y a pas de stockage supplémentaire, on a donc :

$$\mathcal{M}(D_{\mathrm{d}}f) = (n+1) \times \mathcal{M}(f) \tag{4.19}$$

où $D_{\mathrm{d}}f$ désigne l'implémentation de *df* obtenue par l'application du mode direct de différentiation automatique comme exposé en 4.1.2.

Compte tenu de la façon dont on applique les règles de dérivation, le coût additionnel généré par la dérivation des opérations simples est résumé dans le tableau suivant :

	$+$	$\times$	N.L.
$+$	n	0	0
$\times$	n	$2n$	0
N.L.	0	n	1

Ce qui permet de déduire le coût de l'évaluation de la dérivée pour chaque type d'opération :

$$\begin{aligned} \mathrm{ADD}(D_{\mathrm{d}}f) &= (2n+1)\mathrm{ADD}(f) + n\mathrm{MUL}(f) \\ \mathrm{MUL}(D_{\mathrm{d}}f) &= (2n+1)\mathrm{MUL}(f) + n\mathrm{ONL}(f) \\ \mathrm{ONL}(D_{\mathrm{d}}f) &= 2\mathrm{ONL}(f) \end{aligned} \tag{4.20}$$

à comparer par exemple avec le surcoût qu'engendrerait une utilisation de la formule des différences finies centrées, pour laquelle on a vu que l'on aurait $\mathrm{OP}(D_{\mathrm{df}}f) = 2n\mathrm{OP}(f)$, pour tout OP = ADD, MUL ou ONL. La précision machine a donc un prix ! Remarquons que la complexité est ici indépendante du nombre m de variables de sortie à dériver.

Complexité(s) du mode inverse

Dans le cadre du mode inverse, chaque variable se voit adjoindre un vecteur de taille m. On en déduit un premier surcoût en mémoire de $m \times \mathcal{M}(f)$. Les surcoûts correspondant aux opérations directement imputables à la méthode sont les suivants :

	+	×	N.L.
+	$2m$	0	0
×	$2m$	$2m$	0
N.L.	m	m	1

On déduit de ce tableau que le surcoût est dominé par $4m \times \mathcal{C}(f)$, ce qui semble très encourageant pour la dérivation de codes modélisant une fonction à valeurs scalaires. En effet, contrairement au mode direct, la mémoire nécessaire et les calculs additionnels paraissent ne pas du tout dépendre du nombre de paramètres de dérivation. Cette complexité n'est malheureusement valable que dans le cas où le code calculant f est une implémentation directe d'une de ses séquences d'évaluation. La trace d'évaluation, une fois calculée par le programme, est alors toujours accessible contrairement à ce qui se passe lors de l'exécution d'un programme ordinaire dans lequel il est tout à fait normal, et même souhaitable, que des variables soient détruites ou écrasées. On a vu précédemment qu'il existe deux possibilités pour traiter ce problème de manière générique, possibilités qui consistent soit à effectuer une sauvegarde de toutes les valeurs adoptées par chaque variable, soit rappeler le programme à chaque fois pour ré-obtenir les valeurs perdues. Opter pour la première solution permet d'éviter un gros surcoût en calculs au prix d'une augmentation potentiellement colossale de la mémoire nécessaire puisque la transformation d'un code en séquence d'évaluation aura *a priori* besoin d'autant de nouvelles variables que d'utilisations des opérateurs d'affectation :

$$\mathcal{M}(D_{\mathrm{i}} f) = (m+1) \times \mathcal{M}(f) + \alpha \times \mathcal{C}(f)$$

Le nombre d'opérations arithmétiques et d'appels à des fonctions non linéaires ne varie pas dans ce cas. La complexité reste donc de l'ordre de $4m \times \mathcal{C}(f)$. Le temps d'exécution sera néanmoins sensiblement plus long dans la mesure où l'enregistrement des valeurs intermédiaires va se révéler très onéreux.

Le stockage de toutes les valeurs intermédiaires d'une application se révélera dans la plupart des cas problématique. Même pour un calcul modeste dont la durée serait de l'ordre de quelques minutes, la taille de l'enregistrement se mesurerait en Go et il serait donc illusoire de vouloir conserver celui-ci dans la mémoire vive de la machine. On aurait donc à mettre au point un gestionnaire de mémoire virtuelle dont la lecture est, comme chacun sait, relativement lente et donc inappropriée à une utilisation avec accès intensif.

L'alternative offerte par le re-calcul des valeurs intermédiaires économise quant à elle la mémoire, qui reste alors en $(m+1)\mathcal{M}(f)$, mais avec une élévation à la puissance 2 de la complexité arithmétique de l'évaluation de f. Partant de Y_n, pour calculer les composantes de Y_{N-1} (notations de 4.3), on aura besoin de x_{N-1} qui aura peut-être été écrasé lors du calcul de x_N, ou bien encore été détruit. On doit donc refaire les $N-1$ opérations précédant son calcul. Puis on devra refaire à nouveau les $N-2$ opérations nécessaires au calcul de x_{N-2} dont on a besoin pour Y_{N-2}, *etc.* Le nombre d'opérations, tous types confondus, impliquées dans cette stratégie sera donc $\mathcal{N} = N(N-1)/2 \underset{n \gg 1}{\sim} N^2/2$. Une telle explosion du temps de calcul n'est tout simplement pas acceptable pour des applications réelles. Aussi, lorsque cette méthode est utilisée, c'est conjointement à la précédente. On pourra par exemple choisir de re-calculer les variables intermédiaires rencontrées en début de programme que l'on obtiendrait rapidement, et ne stocker que les valeurs nécessitant une exécution plus complète du code. On peut aussi, et c'est en pratique inévitable pour espérer pouvoir calculer un gradient par différentiation automatique inverse, parsemer le programme évaluant f de *checkpoints* (que nous appellerons en français *balises*) en lesquels seront sauvegardées toutes les variables et à partir desquels on lancera le calcul

des suivantes. Si par exemple on répartit Q balises uniformément parmi les N instructions du programme, le coût du re-calcul sera alors :

$$\mathcal{N} = \frac{Q}{2}\left\lfloor\frac{N}{Q}\right\rfloor\left(\left\lfloor\frac{N}{Q}\right\rfloor - 1\right) \underset{n\gg 1}{\sim} \frac{N^2}{2Q}$$

Cette astuce divise donc la complexité arithmétique par un facteur Q. Il est par contre là encore peu probable que l'intégralité des enregistrements puisse être stockée en mémoire vive et il faudra à nouveau passer par un stockage de ceux-ci sur le disque dur. C'est cependant moins problématique ici car les informations nécessaires ne sont lues que lors du passage sur une balise, le calcul des variables intermédiaires jusqu'à la balise suivante pouvant se faire exclusivement à l'aide du cache et de la mémoire vive. Les problèmes d'écriture et de lecture en mémoire de masse réapparaissent si un trop grand nombre de balises est utilisé, puisque qu'une plus grande proportion du temps sera consacrée aux chargements des valeurs enregistrées sur le disque dur qu'au calcul des autres intermédiaires. On a donc là un problème de recherche du nombre optimal de balises pour trouver un équilibre entre lecture en mémoire de masse et coût arithmétique.

Bornes inférieures sur les complexités

Nous venons de donner quelques éléments de réflexion en relation avec la complexité algorithmique des deux modes de différentiation automatique dans leurs versions les plus élémentaires. Des majorants des coûts des versions différentiées des programmes, en fonction de celui de l'implémentation initiale, ont été établis. On peut également établir *au cas par cas*, c'est-à-dire d'une manière plus explicitement dépendante de la séquence d'évaluation initiale, des minorants du surcoût engendré par la différentiation. Les complexités ainsi obtenues correspondraient alors à celles associées à une implémentation optimale du calcul des dérivées du programme initial, implémentation dans laquelle toute opération arithmétique impliquant son élément neutre ou absorbant serait évitée de sorte à ne garder que les calculs ayant une réelle importance.

Considérant 4.9 et 4.12, on voit que de telles opérations vont être très nombreuses à l'amorçage du procédé de différentiation. Plus généralement, dans le cadre des algorithmes naïfs que nous avons présentés, toute instruction *proche* d'une racine dans l'arbre d'évaluation de la différentielle va comporter des opérations inutiles, que l'exploitation d'une plus grande quantité d'informations relatives au programme à différentier pourrait éviter.

Un exemple typique illustrant le gâchis engendré par les méthodes de différentiation automatique est celui du carré scalaire d'un vecteur dont les composantes sont les entrées du programme $s = \sum_{i=-n}^{1} x_i^2$. En mode direct naïf, le programme différentié va calculer $\nabla s = \sum_{i=1}^{n} 2x_{-i}\mathbf{e}_i$, ce qui correspond à $n(n-1)$ multiplications par 0, n multiplications par 1 et $(n-1)^2$ additions impliquant un 0.

En s'inspirant de l'analyse de [46], établissons les notions de *support étendu* et d'*image étendue*. Pour ces définitions, nous allons alternativement assimiler chaque élément de la trace d'évaluation à une fonction de $(x_{-n}, \ldots, x_{-1})$ ou de tous les éléments d'ordres inférieurs de la trace selon les besoins. On appellera alors, et on notera $\mathcal{X}_k$ et $\mathcal{Y}_k$, pour $k \in [\![-n, N]\!]$, respectivement le *support étendu* et l'*image étendue* de l'instruction k (ou de la variable x_k), les ensembles :

$$\mathcal{X}_k = \left\{ i \in [\![1, n]\!] \;\middle|\; \frac{\partial x_k}{\partial x_{-i}} \neq 0 \right\} \qquad \mathcal{Y}_k = \left\{ j \in [\![1, m]\!] \;\middle|\; \frac{\partial x_j}{\partial x_k} \neq 0 \right\} \tag{4.21}$$

On a évidemment $|\mathcal{X}_k| \leq n$, $|\mathcal{Y}_k| \leq m$, ainsi que $\mathcal{X}_{-i} = \{i\}$ pour $1 \leq i \leq n$, $\mathcal{Y}_{N-m+j} = \{j\}$

pour $1 \leq j \leq m$, et des relations de récurrence pour :

$$\forall k \geq 0,\ \mathcal{X}_k = \bigcup_{i \in I_k} \mathcal{X}_i \qquad \forall k \leq N-m,\ \mathcal{Y}_k = \bigcup_{j>k | k \in I_j} \mathcal{Y}_j$$

Cette relation implique la croissance de $|\mathcal{X}_k|$ lorsque l'on remonte une branche de l'arbre associé à la séquence d'évaluation et inversement pour $|\mathcal{Y}_k|$.

Support et image étendus, en plus de donner lieu à l'élaboration de versions plus sophistiquées des différents modes de différentiation automatique (voir section suivante), permettent d'établir le coût de la différentiation d'une séquence d'évaluation par une implémentation optimale du mode correspondant. On suppose pour cela que l'évaluation de la dérivée d'une fonction élémentaire φ_k est de complexité comparable à celle de la primitive et, plus suspect, que le problème de récupération des données est résolu gratuitement :

$$\mathcal{C}\left(D^{\mathrm{d}}_{\mathrm{opt}} f\right) = \sum_{k=0}^{N} (1 + |\mathcal{X}_k|)\, \mathcal{C}\left(\varphi_k\right) \qquad \mathcal{C}\left(D^{\mathrm{i}}_{\mathrm{opt}} f\right) = \sum_{k=0}^{N} (1 + |\mathcal{Y}_k|)\, \mathcal{C}\left(\varphi_k\right)$$

On définit le *support moyen* et l'*image moyenne* :

$$\overline{n} = \frac{\sum_{k=0}^{N} |\mathcal{X}_k| \mathcal{C}\left(\varphi_k\right)}{\sum_{k=0}^{N} \mathcal{C}\left(\varphi_k\right)} \qquad \overline{m} = \frac{\sum_{k=0}^{N} |\mathcal{Y}_k| \mathcal{C}\left(\varphi_k\right)}{\sum_{k=0}^{N} \mathcal{C}\left(\varphi_k\right)} \tag{4.22}$$

En plus de fournir une quantification statistique de la séparabilité de la fonction f, ces indicateurs permettent d'exprimer simplement la complexité optimale théoriquement possible du calcul par différentiation automatique des composantes du gradient f en un point puisque l'on a :

$$\mathcal{C}\left(D^{\mathrm{d}}_{\mathrm{opt}} f\right) = (\overline{n}+1)\mathcal{C}\left(f\right) \qquad \mathcal{C}\left(D^{\mathrm{i}}_{\mathrm{opt}} f\right) = (\overline{m}+1)\mathcal{C}\left(f\right) \tag{4.23}$$

Pour certaines fonctions pathologiques, le support moyen (ou son homologue adjoint $\overline{m}$) ne dépend pas de la dimension n de l'espace de définition de f. Si par exemple $f : \mathbb{R}^n \to \mathbb{R}^m$ est telle que chacune de ses composantes est le prolongement constant à $\mathbb{R}^n$ d'une fonction de $\mathbb{R}^p$ dans $\mathbb{R}$, avec $p \leq n$ (et que chaque composante du gradient de chacune de ces m fonctions n'est pas identiquement nulle), on a alors $\overline{n} = p$. Dans le cas $m = 1$, où f est à valeurs dans $\mathbb{R}$, si on peut écrire cette dernière comme la somme ou le produit de fonctions du type précédemment défini, on aura encore $\overline{n} \simeq p$ (selon la complexité des termes), $p = 1$ correspond alors au cas extrême où f est dite additivement (ou multiplicativement) séparable :

$$\forall x \in \mathbb{R}^n,\ f(x) = \sum_{i=1}^{n} f_i(x_i) \quad \text{ou} \quad \prod_{i=1}^{n} f_i(x_i)$$

En effet dans ce cas-là, en dehors des n additions ou multiplications finales $\mathcal{X}_k = 1$, on a : $\overline{n} \times \mathcal{C}\left(f\right) = \sum_{i=1}^{n} \mathcal{C}\left(f_i\right) + \sum_{i=1}^{n-1} (i+1)\mathcal{C}\left(+/\times\right)$. En supposant le coût d'une addition ou multiplication égal à 1, cela mène à $\overline{n} \times \mathcal{C}\left(f\right) = \mathcal{C}\left(f\right) + \sum_{i=1}^{n-1} i$, d'où $\overline{n} = 1 + \frac{n(n-1)}{2\mathcal{C}(f)}$. Le coût d'une fonction séparable est en général proportionnel (au moins asymptotiquement) à n, $\mathcal{C}\left(f\right) \sim an$, et on a alors $\overline{n} \sim n/2a$. Mais en pratique, le facteur de proportionnalité a est plutôt grand, de sorte que le comportement asymptotique est peu observé et, à moins que les termes soient très simples à calculer, on observera davantage le comportement

asymptotique selon a : $\overline{n} \underset{a\to+\infty}{\sim} 1 + \frac{n-1}{2a} \sim 1$. Cela signifie que, au regard de 4.23, une implémentation optimale du calcul par différentiation automatique du gradient en un point d'une fonction séparable ne devrait pas dépendre du nombre de paramètres de différentiation, sauf quand ce dernier devient très grand (par rapport à la complexité de f), et ne devrait pas nécessiter un nombre d'opérations élémentaires supérieur à celui nécessaire au calcul de f.

Il n'y a pas de prédiction possible pour le cas général. On peut toutefois donner le résultat trivial $\overline{n} = n/2$ pour une séquence d'évaluation au long de laquelle les supports croissent linéairement. Nous présentons dans la section 4.3 des calculs de supports moyens pour certains calculs *standard* fréquemment rencontrés dans le cadre de la résolution d'EDP ou de problèmes de minimisation basés sur des méthodes d'éléments finis. On dégagera quelques tendances en fonction des types de problèmes considérés.

4.1.6 Vers de meilleurs performances

Bien qu'il soit difficile d'atteindre les valeurs théoriques minimales des complexités, on pourra facilement peaufiner les méthodes naïves que nous avons exposées. Il serait par exemple avantageux de pouvoir distinguer les variables susceptibles d'être traitées en tant que constantes par rapport au procédé de différentiation, et ainsi faire l'économie d'un calcul dont on connaît le résultat. Dans le cadre du mode inverse, il est par ailleurs inutile de sauvegarder ou de recalculer les valeurs intermédiaires issues de calculs n'impliquant que l'addition et la soustraction. Ces deux précautions constituent un premier pas dans l'élimination des calculs superflus.

Au-delà de ces quelques menues améliorations, il n'existe que peu d'opportunités de réduction des coûts pour le mode direct. Citons le travail de [6] qui exploite le concept d'*expression template* développé dans [1]. Cette technique, dont l'application est restreinte au cas de la différentiation automatique en mode direct par *surcharge d'opérateurs* en `C++` (operator overloading - voir section 4.2), permet d'éviter certains coûts de manipulation mémoire induits par les spécificités des anciens standards `C++` (le nouveau standard, `C++-11`, comporte cependant des outils qui rendent caduque l'exploitation de cette technique à ces fins puisque un résultat équivalent pourra être obtenu, et ce bien plus facilement, par l'utilisation des *move constructor* et des *rvalue references*). Le gain est toutefois discutable et n'est réellement significatif que lorsque le programme différentié comporte des instructions impliquant de grandes expressions mathématiques. Il ne s'agit donc pas ici d'essayer d'éliminer les opérations stériles mentionnées précédemment, mais uniquement de tempérer des surcoûts introduits par les standards de programmation. Plus récemment [78] propose un raffinement de ce thème où le programme est globalement différentié en mode direct par surcharge d'opérateurs, mais les expressions mathématiques locales le sont en mode inverse. Mais là encore, cette méthode, bien que très astucieuse, n'apporte de réelles améliorations que dans le cas où le code différentié comporte un grand nombre de calculs avec de longues expressions.

La question de l'utilisation de vecteurs creux pour le mode direct a été étudiée dans [91]. Cette stratégie semble être le seul moyen pour se rapprocher significativement de la complexité minimale dans le cadre du mode direct par surcharge des opérateurs. On est alors confronté à deux nouveaux problèmes : la question du stockage compact des données, qui est déjà ancienne et constitue un domaine de recherche par elle-même, et la réunion efficace des domaines à chaque opération. Dans l'état actuel, le surcoût généré peut se montrer supérieur à l'économie réalisée sur le calcul des dérivées si les supports augmentent rapidement. Aussi, cette méthode reste réservée à la différentiation de fonctions à faible

support moyen.

On trouve une littérature plus abondante concernant l'amélioration du mode inverse. Beaucoup d'efforts sont déployés pour essayer de juguler l'énorme surcoût que ce mode engendre, car il est le seul à même de fournir des résultats dans le cas d'un nombre élevé de paramètres de différentiation. Il est par ailleurs inévitable pour le calcul complet de matrices hessiennes, même quand le nombre de paramètres reste modérément élevé, un tel calcul en mode direct pur n'étant possible que pour des hessiennes de taille très réduite.

On a précédemment vu que seule l'association balisage/re-calcul permettait à ce mode d'être viable. Il apparaît que le balisage uniforme que nous avons exposé n'est cependant pas optimal, et la question de déterminer une répartition optimale des balises reste un problème ouvert et semble relever du domaine des problèmes NP-complets (voir [74]). On trouvera dans [25] un inventaire assez synthétique des diverses stratégies élaborées à ce jour.

Pour les deux modes de différentiation automatique, une place importante est faite à l'analyse statique du programme source pour essayer de trouver un programme différentié optimal (du point de vue des opérations arithmétiques). Cette étude statique du graphe de la fonction implémentée permet par ailleurs d'établir *a priori* diverses caractéristiques de la différentielle pouvant s'avérer plus ou moins précieuses (structure des matrices creuses, remplissage au cours de l'accumulation des dérivées, parallélisme, *etc.*). Ces problèmes faisant appel à des éléments de théorie des graphes qui s'éloignent de notre centre d'intérêt, nous ne les aborderons pas davantage dans cette thèse. Il est tout de même intéressant de noter que, comme il est affirmé dans [46] et [74], la recherche du calcul du gradient à moindre coût par différentiation automatique est très probablement un problème NP-complet, comme c'est souvent le cas lorsqu'une question bascule dans le domaine de l'analyse combinatoire.

4.2 Implémentation

Il existe deux stratégies pour la mise en œuvre de chacun des modes de différentiation automatique. Il s'agit de la *transformation de sources* et de la *surcharge d'opérateurs*. La première consiste à écrire un programme qui prend un autre programme comme argument pour en donner une version différentiée, qu'il faut alors éventuellement compiler. L'avantage de cette méthode est de pouvoir se pratiquer avec n'importe quel langage de programmation et, surtout, de présenter un potentiel d'optimisation du résultat supérieur à la surcharge d'opérateurs. La principale difficulté soulevée par la transformation de sources est la complexité de son implémentation, qui nécessite implicitement la programmation d'outils d'analyse syntaxique. Le développement d'un outil de différentiation exploitant la surcharge d'opérateurs est quant à lui beaucoup plus simple, avec une portée didactique certaine. Cela n'est néanmoins possible qu'avec des langages de programmation supportant cette fonctionnalité, et le programme obtenu est souvent moins efficace qu'une version différentiée conçue par transformation de sources, généralement à cause d'opérations arithmétiques ou de lectures et écritures en mémoire supplémentaires induites par la méthode.

Une liste exhaustive des différents outils de différentiation actuellement disponibles et d'articles scientifiques qui leur sont consacrés est disponible sur le site [2], ou éventuellement sur l'article *Automatic Differentiation* de l'encyclopédie collaborative Wikipédia. Contentons nous ici de citer les ténors du genre. TAPENADE [48] développé par l'équipe TROPICS de l'INRIA Sophia-Antipolis, ainsi que le compilateur `dcc` [75] du laboratoire d'informatique de l'université d'Aix-la-Chapelle (Aachen), sont deux ou-

tils performants de transformation de sources Fortran, C et C++ , le second présentant la particularité de compiler automatiquement le fichier source obtenu. Pour ce qui est de la surcharge d'opérateurs, nous mentionnerons ADOL-C [94], une bibliothèque open-source très complète, développée et maintenue dans le cadre du COIN-OR project (`http://www.coin-or.org/`), dont il a déjà été question dans ce manuscrit.

Nous donnons ici un aperçu technique de la mise en œuvre de la différentiation automatique en mode direct. Les exemples sont écrits dans le langage C++ qui est, à notre avis, le langage supportant la surcharge d'opérateurs le plus largement connu. Un lecteur non familiarisé avec ce langage trouvera toutes les explications nécessaires dans l'ouvrage de son créateur [88]. On omettra parfois les instructions de préprocesseur du type `#include` pour des en-têtes de la bibliothèque standard (STL), ou encore les instructions `using` tout en faisant fi de l'opérateur de résolution de portée `::` qui permet habituellement d'utiliser des objets définis dans des espaces de noms nommés, ceci, parce que nous nourrissons quelques scrupules quant à la clarté de ce document et qu'un morceau de code C++ peut facilement prendre une apparence cryptique.

4.2.1 Implémentation d'un outil de transformation de sources

Le principe, comme nous l'avons déjà entrevu, est de concevoir un programme qui prendra en argument les programmes que l'on souhaite différentier et écrira à partir de celui-ci le programme implémentant sa "différentielle". Il restera alors à compiler le code obtenu pour obtenir l'exécutable calculant effectivement les quantités souhaitées.

La transformation de sources procéderait comme le ferait un programmeur qui différentierait le code manuellement par une application directe de la loi de dérivation des fonctions composées. Tout comme ce programmeur, un outil de différentiation de sources à sources puissant doit être capable de repérer les variables indépendantes de celles par rapport auxquelles le code doit être différentié, dans un souci de réduction des calculs. Mais ce paradigme de différentiation automatique laisse aussi le champ libre à une multitude d'opportunités en matière d'optimisation du code à la compilation. C'est pourquoi ce type d'outil de différentiation produit en général du code beaucoup plus efficace que celui généré par la surcharge d'opérateurs. Cela est d'autant plus vrai dans le cadre du mode inverse pour lequel la surcharge d'opérateur, si elle constitue un élégant exercice de programmation avancée, se révèle un choix désastreux dans la pratique, où seuls les outils de transformation de sources procurent une solution convaincante.

L'implémentation d'un outil de transformation de sources ne diffère de la méthode par surcharge d'opérateurs que par le fait que l'analyse syntaxique et la génération du code différentié sont explicitement écrites par le programmeur. En surcharge d'opérateurs, ces étapes sont effectuées par le compilateur. D'un point de vue purement conceptuel, il n'y a donc pas vraiment lieu d'opérer de dichotomie entre les deux paradigmes, c'est pourquoi nous nous bornerons à seulement expliciter en détail l'implémentation d'un outil de différentiation par surcharge d'opérateurs, afin de nous concentrer sur des points de programmation dont les rapports avec la dérivation restent manifestes et éviter la prolifération de détails techniques.

4.2.2 Implémentation d'une bibliothèque de différentiation automatique en mode direct par surcharge d'opérateurs

La tâche est dans ce cas beaucoup plus simple et constitue d'ailleurs un exercice de programmation aux qualités didactiques indéniables, permettant d'aborder à la fois les notions de programmation orientée objets, la surcharge d'opérateurs et éventuellement la

programmation générique (template). Nous abordons dans cette courte section un point de vue académique, dans le sens où nous mettons directement en œuvre les principes de la section théorique précédente. Les questions de performances ne dépasseront que rarement les principes généraux de programmation efficace en C++ (dans quel cas passer les arguments par références constantes, non constantes, par valeurs, utilisation ou non de l'héritage, *etc.*) que nous supposerons connus du lecteur (que nous renvoyons aux ouvrages généraux sur le C++ précédemment cités dans le cas contraire).

La différentiation automatique par surcharge d'opérateur consiste à remplacer toutes les variables du type numérique `double` (ou `float`) par des variables d'un nouveau type contenant, en plus de la variable remplacée, les membres nécessaires au calcul des dérivées selon l'un des modes décrit dans les sections précédentes. Tous les opérateurs arithmétiques et toutes les fonctions mathématiques impliquant le type `double` sont redéfinis par surcharge afin que, en plus de calculer l'opération qu'ils implémentent, ils calculent les dérivées de cette opération par rapport à certaines variables du programme. Ainsi, si notre nouveau type est `ddouble`, il suffirait *a priori* de remplacer toutes les occurrences de `double` dans le code que l'on désire différentier par `ddouble`.

Surcharge d'opérateurs en mode direct

On suppose que le nombre maximal de paramètres de différentiation n est connu à la compilation, et stocké dans une variable globale `MaxNumberOfDerivatives`. Cela évitera de surcharger la définition des constructeurs et permet d'utiliser le constructeur et l'opérateur de copie par défaut. On définit une nouvelle classe `ddouble` encapsulant la valeur d'une variable flottante (ici en double précision) et de ses "dérivées". On la munit de quelques accesseurs et d'un opérateur d'affectation d'une valeur constante de type `double`. L'argument `init_dx` permet de contrôler la boucle d'initialisation du tableau des dérivées qui est inutile dans certains cas (voir la définition des opérateurs binaires et des fonctions).

```
1  class ddouble {
2    public:
3      static const int N = MaxNumberOfDerivatives;
4      static int nd; //nombre de dérivé courant
5    private:
6      double x;
7      double dx[N];
8    public:
9      //Declaration et initialisation d'un ddouble par une constante
10     ddouble(double c=0,bool init_dx=true) : x(c)
11     {if(init_dx) for(int i=0;i<nd;++i) dx[i] = 0;}
12     // accesseurs :
13     const double val() const {return x;}
14     const double d(int i) const {return dx[i];}
15     double& val() {return x;}
16     double& d(int i) {return dx[i];}
17     //affectation d'une valeur constante :
18     ddouble& operator=(double c)
19     {
20       x=c;
21       for(int i=0;i<nd;++i) dx[i] = 0;
22       return *this;
```

```
23      }
```

On définit également quelques opérateurs couplant opération arithmétique et affectation. On pourrait économiser la définition de ce genre d'opérateurs lorsqu'ils sont appelés avec un argument de type `double`, puisque le constructeur en ligne 10 définit une conversion implicite vers notre type `ddouble`. Il est cependant absurde de créer une variable temporaire de ce dernier type, avec le membre `dx` rempli de zéros, pour ensuite procéder à de multiples additions de variables nulles. Remarquons que pour les opérateurs associés à la division et la multiplication, dont les valeurs des membres `x` sont utilisées dans le calcul des dérivées, il est important de d'abord effectuer la mise à jour du contenu du tableau `dx` puis de procéder à celle de `x`.

```
24      //opérateurs arithmétiques d'affectation
25      //(f+c)' = f' si c constante
26      ddouble& operator+=(double c) {x += c; return *this;}
27      ddouble& operator+=(const ddouble &X)
28      {
29        for(int i=0;i<nd;++i) dx[i] += X.dx[i];//(f+g)' = f' + g'
30        x += X.x;
31        return *this;
32      }
33      ddouble& operator*=(double c)
34      {
35        for(int i=0;i<nd;++i) dx[i] *= c;//(cf)'=cf' si c constante
36        x *= c;
37        return *this;
38      }
39      ddouble& operator*=(const ddouble &X)
40      {
41        for(int i=0;i<nd;++i) dx[i] = dx*X.x + x*X.dx[i];
42        x *= X.x;
43        return *this;
44      }
45      //déclarations et définitions semblables pour les opérateurs
46      //membres -= et /=
47      ...
```

Définissons enfin une fonction membre `SetDiff(int i)` qui permettra de déclarer l'instance de `ddouble` sur laquelle elle est appelée comme le `i`-ème paramètre par rapport auquel seront calculées les dérivées. La fonction statique `SetDiffNumber(int)` est quant à elle utilisée pour changer le nombre de ces paramètres. Dans une logique de pseudo-gestion des exceptions, elle renvoie la valeur `true` si le nouveau nombre de dérivées est accepté, `false` dans le cas contraire (si l'argument est négatif ou plus grand que `MaxNumberOfDerivatives`). Cela termine la définition de la classe :

```
48      ddouble& SetDiff(int i)
49      {
50        for(int j=0;j<nd;++j) dx[j] = (i==j);
51        return *this;
52      }
```

```
53      static const bool SetDiffNumber(int NewNumberOfDerivatives)
54      {
55        bool ret=true;
56        if(NewNumberOfDerivatives<0) {ret=false; nd=0;}
57        else if(NewNumberOfDerivatives>N) {ret=false; nd=N;}
58        else nd=NewNumberOfDerivatives;
59        return ret;
60      }
61  };
```

Pour être opérationnelle, notre classe n'a plus besoin que de pouvoir être appelée avec les opérateurs binaires, booléens et les fonctions usuelles. La surcharge des fonctions ne présente pas de difficulté particulière. La question de les spécifier comme des fonctions *inline* peut se poser, et compte tenu de la courte longueur de la définition il serait *a priori* raisonnable de le faire :

```
62  //Surcharge de la fonction racine carrée
63  inline ddouble sqrt(const ddouble &X)
64  {
65    const double sqrtx = sqrt(X.val());
66    ddouble Y(sqrtx,false);//inutile d'initialiser dx
67    for(int i=0;i<ddouble::nd;++i) Y.d(i) = X.d(i)/(2.*sqrtx);
68    return Y;
69  }
70  //Surcharge de la fonction cosinus
71  inline ddouble cos(const ddouble &X)
72  {
73    ddouble Y( cos(X.val()),false);
74    const double msinx = -sin(X.val());
75    for(int i=0;i<ddouble::nd;++i) Y.d(i) = X.d(i)*msinx;
76    return Y;
77  }
78  //Déclarations et définitions semblables pour toute autre fonction
79  //mathématique de cmath
80  ...
```

La surcharge des opérateurs binaires est à peine plus compliquée. La seule difficulté réside dans le choix ou non de redéfinir ceux-ci avec des arguments non uniformes, autrement dit avec des prototypes tels que `ddouble(ddouble,double)` et `ddouble(double,ddouble)`. Ces surcharges additionnelles sont indispensables pour générer des codes aux performances acceptables, mais sont susceptibles de générer des ambiguïtés si la classe `ddouble` comporte des opérateurs de conversion implicite. C'est d'ailleurs la raison pour laquelle nous n'en n'avons pas déclaré dans notre classe `ddouble` bien que cela soit tentant. Dans tous les cas, comme il est expliqué dans [70], les opérateurs de conversion implicite sont dangereux et mieux vaut les éviter comme la peste. Voici nos deux opérateurs binaires favoris, surchargés pour accepter de travailler avec des `ddouble` :

```
81  //surcharge de l'addition
82  inline ddouble operator+(const ddouble &l,const ddouble &r)
83  {
84    ddouble lpr(l.val()+r.val(),false);
```

```
85   for(int i=0;i<ddouble::nd;++i) lpr.d(i) = l.d(i)+r.dx(i);
86   return lpr;
87 }
88 //addition avec prototype non uniforme
89 inline ddouble operator+(const ddouble &l,double r)
90 {
91   ddouble lpr(l.val()+r,false);
92   for(int i=0;i<ddouble::nd;++i) lpr.d(i) = l.d(i);
93   return lpr;
94 }
95 ... //déclaration et définition semblable de
96     //ddouble operator+(double,const ddouble &)
97
98 //surcharge de la multiplication
99 inline ddouble operator*(const ddouble &l,const ddouble &r)
100 {
101   ddouble lpr(l.val()*r.val(),false);
102   for(int i=0;i<ddouble::nd;++i)
103     lpr.d(i) = l.d(i)*r.val()+l.val()*r.d(i);
104   return lpr;
105 }
106 //multiplication avec prototype non uniforme
107 inline ddouble operator*(const ddouble &l,double r)
108 {
109   ddouble lpr(l.val()*r,false);
110   for(int i=0;i<ddouble::nd;++i) lpr.d(i) = l.d(i)*r;
111   return lpr;
112 }
113 ... //déclaration et définition semblable de ddouble operator*(double,
      const ddouble &), ainsi que pour les opérateurs de division et de
      soustraction
```

Il existe aussi une poignée de fonctions de deux variables qu'il faudrait surcharger de façon similaire aux opérateurs ci-dessus. Il s'agit des fonctions `pow`, `atan2`, *etc.* La redéfinition est similaire et nous ne les écrirons pas. La surcharge des opérateurs booléens (==, !=, <, <=,*etc...*) revient à renvoyer le résultat de la même opération appliquée aux membres `x` des arguments :

```
114 inline bool operator==(const ddouble &l,const ddouble &r)
115 {return l.val()==r.val();}
116 inline bool operator>=(const ddouble &l,const ddouble &r)
117 {return l.val()>=r.val();}
118 ...
```

avec, ici aussi, quelques surcharges pour des types d'arguments non uniformes, afin d'éviter les conversions de `double` vers `ddouble`, très coûteuses relativement à l'opération initialement visée. Pour ces opérateurs, les fonctions sont à déclarer `inline` sans hésitation.

Ces lignes de code constituent le minimum vital pour pratiquer la différentiation automatique en mode direct par surcharge d'opérateurs. Différentier avec ces classes n'est pas compliqué, surtout si le programme utilisant ces classes n'est pas encore écrit : il suffira de déclarer de type `ddouble` toute variable dont on sait qu'elle dépend des paramètres par

rapport auxquels on veut dériver, laisser le type `double` habituel aux variables jouant le rôle de *constantes* d'un point de vue mathématique, et enfin appeler la méthode `SetDiff` sur les variables par rapport auxquelles les dérivées seront calculées. Le programme généré peut de plus être utilisé pour l'évaluation de la fonction primitive : il suffirait pour cela d'affecter la valeur 0 à `NumberOfDerivatives`. La modification de sources déjà existante est par contre plus délicate, la tentation étant grande de remplacer toute occurrence de `double` par `ddouble`. Ce ne serait pas dramatique, mais un temps de calcul non négligeable serait alors alloué au calcul des membres `dx` de variables mathématiquement constantes par rapport aux entrées, ce qu'un outil de transformation de sources devrait en revanche détecter pour en conséquence. On retiendra donc que la transformation de sources est préférable pour la différentiation de codes anciens, surtout lorsque ceux-ci sont longs et qu'une analyse manuelle se révélerait laborieuse.

Utilisation

Abandonnons nos exemples triviaux pour donner un petit aperçu de la manière dont cet embryon de bibliothèque de différentiation automatique s'utilise. Supposons que, étant donnés une matrice carrée A de taille $n \in \mathbb{N}$, et un vecteur $b \in \mathbb{R}^n$, on souhaite implémenter la fonction $x \in \mathbb{R}^n \mapsto (Ax, x) - (b, x)$ ainsi que son gradient. On suppose que l'on dispose pour cela d'un modèle de classe `matrix` paramétré par le type de coefficients, muni de l'accesseur `operator()(int,int)` écrit dans le tout dernier standard `C++-11` ([89]) de sorte à ce que l'on puisse initialiser une matrice par une liste d'initialiseurs. Pour une utilisation la plus simple possible, on ne devrait définir que des matrices et des vecteurs dont le type sous-jacent est `ddouble`. Nous allons cependant tirer profit des techniques de méta-programmation qui ont été ajoutées dans le dernier standard du `C++` afin d'éviter les calculs inutiles.

```
1  #include <type_traits> //l'en-tête pour obtenir des informations sur
       les types à la compilation
2  #include <vector> //le vecteur de la STL
3  #include "ddouble.hpp" //notre bibliothèque
4  #include "matrix.hpp" //les matrices (pas dans la STL)
5
6  struct unknown_ddouble_cast_type {};
7  template<typename T,bool F=is_fundamental<T>::value>
8  struct ddouble_cast_struct {
9    typedef unknown_ddouble_cast_type type;
10 };
11 template<typename T> struct ddouble_cast_struct<T,true> {
12   typedef ddouble type;
13 };
14 template<bool F> struct ddouble_cast_struct<long double,F> {
15   typedef unknown_ddouble_cast_type type;
16 };
17
18 namespace std{
19   template<typename T> class common_type<ddouble,T>
20   {typedef typename ddouble_cast_struct<T>::type type;};
21   template<typename T> class common_type<T,ddouble>
22   {typedef typename ddouble_cast_struct<T>::type type;};
23 }
24
```

```
25 using namespace std;//pour se libérer du préfixe std
```

Le type `common_type<T1,T2,...,TN>::type` correspond au type `Ti` vers lequel tous les `Tj` supportent la conversion sans perte d'information. Les étranges lignes de codes ci-dessus permettent d'étendre à notre nouveau type le modèle de classe `common_type` : si `T` est un type fondamental (`int`, `double`, `float`, *etc...*), alors une opération mélangeant `ddouble` et `T` doit renvoyer un `ddouble`, avec exception pour le `long double` puisque la conversion de celui-ci vers le type sous-jacent à nos `ddouble` comporte une perte d'information. Dans les autres cas on renvoie un type d'erreur qui fera échouer la compilation. On trouvera dans l'annexe C.1 une spécialisation plus complète de `common_type` dans le cas où `ddouble` est paramétré par son type sous-jacent. Munis de ces outils de détermination automatique du type, le produit matrice-vecteur et le produit scalaire se programment comme suit :

```
26 //produit matrice-vecteur
27 template<typename MT,typename VT>
28  vector<typename common_type<MT,VT>::type>
29 operator*(const matrix<MT> &M,const vector<VT> &v)
30 {
31   assert(M.m==v.size());
32    vector<typename common_type<MT,VT>::type> X(M.n,0.);
33   for(int i=0;i<M.n;++i)
34     for(int j=0;j<M.n;++j) X[i] += M(i,j)*v[j];
35   return X;
36 }
37
38 //produit scalaire
39 template<typename T1,typename T2>
40 typename common_type<T1,T2>::type
41 operator,(const vector<T> &u,const vector<T> &v)
42 {
43   assert(u.size()==v.size());
44   typename common_type<T1,T2>::type ps=0;
45   for(int i=0;i<u.size();++i) ps += u[i]*v[i];
46   return ps;
47 }
```

Les structures de données représentant la matrice A et le vecteur b ne devront contenir que des `double` puisque d'un point de vue mathématique, il s'agit de constantes. Seuls les arguments de la fonction `C++` implémentant f seront de type `ddouble` , ainsi que certaines variables intermédiaires nécessaires à l'évaluation de cette dernière. L'intérêt de `common_type` est que la détermination du type est désormais laissée à la discrétion du compilateur (au prix d'une écriture un peu lourde il est vrai, mais cette lourdeur est justement fort appréciée des adeptes du `C++`) :

```
48 //Définitions de A et b. Les valeurs n'ont rien de particulier
49 const matrix<double> A={{ 1,-2,  3,-4},
50                          { 5,-6,  7,-8},
51                          { 9,-10,-9, 8},
52                          {-7, 6, -5, 4}};
```

```
53 const vector<double> b={7,-3,4,5};
54
55 //La fonction :
56 ddouble f(const ddouble &X) {return ((A*X),X) - (b,X);}
```

Passons finalement à l'utilisation dans le `main` de tous nos objets. La fonction membre `matrix::solve(vector b)` résout le système linéaire $Ax = b$. On suppose de plus que sont implémentées les opérations vectorielles standard telles que l'addition et la soustraction entre `vector<T>` pour tous `T` (attention, ce n'est pas le cas dans la STL : il faudrait pour cela utiliser le vecteur `val_array`). Remarquons que l'opérateur de flux << n'est pas surchargé pour fonctionner avec des objets de type `vector`, mais nous supposons ici que c'est le cas :

```
57 //Une version de SetDiff pour tout un vecteur
58 vector<ddouble>& SetDiff( vector<ddouble> &X) {
59   for(int i=0;i<ddouble::nd;++i) X[i].SetDiff(i);
60   return X;
61 }
62
63 typedef vector<ddouble> dvector;
64
65 int main()
66 {
67   dvector X1={0,0,0,0},X2=A.solve(b),
68           X3={1,2,3,4};
69   vector<double> x3={1,2,3,4};
70   cout << "(f(X1) ; grad(f)(X1)) = " << f(SetDiff(X1)) << endl <<
       endl;
71   cout << "(f(X2) ; grad(f)(X2)) = " << f(SetDiff(X2)) << endl <<
       endl;
72   cout << "(f(X3) ; grad(f)(X3)) = " << f(SetDiff(X3)) << endl;
73   cout << "  devrait etre egal à : (" << (((A*x3),x3) - (b,x3));
74   cout << " ; " << (A*x3 - b) << endl;
75   return 0;
76 }
```

Une généralisation

Une approche intéressante consiste à définir la classe `ddouble` en tant que modèle paramétré par le type des données des membres :

```
1 template<typename T> class ddouble {
2   private:
3     T x;
4     T dx[N];
5   public:
6     typedef T value_type;
7     //les autres déclarations et définitions
8     //adaptées au paramètre de patron
9     ...
```

```
};
```

La surcharge des fonctions à une unique variable n'en est pas trop affectée. Les opérateurs binaires et fonctions à deux variables nécessitent un peu de code supplémentaire pour la gestion des types. On peut par exemple aussi utiliser pour cela le modèle de classe `common_type` en adaptant les spécialisations proposées dans la section précédente :

```
namespace std
{
  template<typename T,typename K> class common_type<ddouble<T>,K>
  {typedef typename ddouble<common_type<T,K>::type> type;};
  template<typename T,typename K> class common_type<K,ddouble<T>>
  {typedef typename ddouble<common_type<K,T>::type> type;};
  template<typename T1,typename T2>
  class common_type<ddouble<T1>,ddouble<T2>>
  {typedef typename ddouble<common_type<T1,T2>::type> type;};
}
```

La surcharge de l'opérateur d'addition peut alors s'écrire comme :

```
//Les surcharges de l'addition
template<class L,class R> typename common_type<ddouble<L>,R>::type
operator+(const ddouble<L> &l,const R &r)
{
  typename common_type<ddouble<L>,R>::type res(l);
  res += r;
  return res;
}
template<class L,class R> typename common_type<L,ddouble<R>>::type
operator+(const L &l,const ddouble<R> &r)
{
  typename common_type<ddouble<L>,R>::type res(r);
  res += l;
  return res;
}
template<class L,class R> ddouble<typename common_type<L,R>::type>
operator+(const ddouble<L> &l,const ddouble<L> &l)
{
  typedef ddouble<typename common_type<L,R>::type> return_type;
  return_type res(l.val()+r.val(),false);
  for(int i=0;i<return_type::nd;++i) res.d(i) = l.d(i)+r.d(i);
  return res;
}
```

Un listing détaillé des objets à définir dans ce cadre est proposé dans l'annexe C.1. Une telle démarche ouvre de nombreuses possibilités telles que le calcul de matrices hessiennes, l'intrication de plusieurs dérivations afin par exemple de différentier le résultat de la résolution d'un système non linéaire par la méthode de Newton, l'application de l'algèbre de différentiation à toutes sortes de types pourvu que les opérateurs arithmétiques et les fonctions standard puissent être définis pour le type en question, *etc.* Les possibilités sont

nombreuses et offrent un vaste champ d'investigation pour l'enseignement de la programmation. Une telle intrication de ces structures de données est en revanche dangereuse pour les performances, notamment de par la redondance de données et de calculs qu'elle induit. Une utilisation efficace requiert donc souvent une spécialisation.

4.3 Différentiation automatique dans FreeFem++

Penchons nous désormais sur le cas de FreeFem++. Nous avons déjà évoqué dans la section introductive de ce chapitre les possibilités en termes de développement d'outils pour le calcul automatique de dérivées, avec une préférence pour l'introduction d'outils dont l'utilisation serait proche de celle du mode direct de différentiation automatique par surcharge d'opérateur. Comme le langage de FreeFem++ ne permet pas la définition de classes, de tels outils doivent être directement implémentés dans le noyau du logiciel.

4.3.1 Cahier des charges

Il s'agit donc de fournir à l'utilisateur un nouveau type numérique `treal`, pouvant être passé en tant qu'argument à toute fonction, opérateur ou structure de donnée paramétrable par un type, supportant le type `real`. A l'origine, la volonté était d'appeler ce type `dreal`, mais avec l'idée ambitieuse d'également développer des outils pour la différentiation en mode inverse, il fallait anticiper les ambiguïtés. Le mode direct étant parfois appelé mode *tangent*, le choix s'est porté sur le mot clé `treal` avec le projet d'appeler `areal` le type associé à la différentiation en mode inverse, qui lui se voit parfois appeler mode *adjoint*. Ce nouveau type doit permettre de calculer, en plus du résultat numérique usuel que chacun de ces objets fournit avec le type `real`, l'évaluation de chacune des dérivées partielles de ces résultats par rapport à certaines variables que l'utilisateur doit indiquer. La manière d'accéder au résultat usuel, ainsi qu'aux composantes de ces gradients doit être la plus naturelle possible, et le procédé par lequel les variables de différentiation sont indiquées doit être le plus simple et intuitif possible. Enfin, comme il s'agit d'un outil destiné à être utilisé dans le cadre de simulations numériques, l'efficacité est bien entendue une exigence incontournable.

Muni de ce type `treal`, l'utilisateur peut alors calculer simultanément le résultat d'une fonction et de son gradient par rapport à ses arguments, sans en modifier le code (en dehors du remplacement de chaque occurrence du type `real` par le type `treal`. Le type `treal` peut également être utilisé en dehors de la définition d'une fonction afin de calculer des dérivées à tout moment.

4.3.2 FreeFem++-ad

Phases du développement

L'implémentation de ces fonctionnalités par le mode inverse est relativement difficile, aussi était-il plus prudent d'envisager en premier lieu une approche par le mode direct. Dans cette perspective, nous avons mené une étude de faisabilité (stage au printemps 2009). L'essai consistait à remplacer brutalement le type `double` dans le code `C++` du logiciel (versions 3.1 à 3.3) par la classe `ddouble` précédemment présentée. Le prototype ainsi obtenu était très stable, ce qui nous a encouragé à poursuivre le développement des outils de différentiation de manière plus subtile (il fallait pouvoir garder l'accès aux anciennes fonctionnalités, tous les utilisateurs n'ayant pas besoin de la différentiation automatique).

La possibilité d'utiliser une allocation dynamique du tableau des dérivées a également été testée sur ce prototype. Les pertes de performance, et dans une moindre mesure quelques problèmes de stabilité, nous ont vite amené à reconsidérer cette option. Le meilleur compromis, dans un premier temps, était d'utiliser des tableaux de taille fixe tout en pouvant éventuellement changer le nombre de dérivées réellement utilisées, quitte à gaspiller parfois un peu de mémoire. Le listing de la classe pourra être consulté dans l'annexe C.2.1. L'intégralité du code FreeFem++ a nécessité une longue révision pour pouvoir supporter le nouveau type et de nombreuses modifications ont du y être apportées. Ce long travail, très formateur dans tous les sens du terme, a abouti à une version *alpha* que nous appellerons FreeFem++-ad.

Le cas des solveurs linéaires provenant de bibliothèques

FreeFem++ contient ses propres implémentations du gradient conjugué et de GMRES qui sont paramétrées par le type numérique des données. Ces deux méthodes de résolution de systèmes linéaires ne posent donc pas de problème pour ce qui est du passage à la différentiation automatique. C'est en revanche plus problématique pour les bibliothèques telles que UMFPACK. Il n'aurait pas été raisonnable de modifier celles-ci afin que les types et les opérateurs définis pour la différentiation automatique puissent y être introduits. Nous avons donc imaginé une petite astuce.

Supposons que l'on se retrouve avec une matrice $\mathbf{A}$ et un second membre $\mathbf{b}$ à coefficients `ddouble` avec n dérivées, et que l'on désire calculer le vecteur $\mathbf{x}$ tel que $\mathbf{Ax} = \mathbf{b}$ avec différentiation automatique. Cela signifie que l'on dispose en réalité d'une matrice A stockant les valeurs de $\mathbf{A}$, et de n matrices A_i contenant les valeurs de $\partial A/\partial v_i$ (où les v_i sont les variables de dérivation), de même que $\mathbf{b}$ est la donnée d'un vecteur b et de n vecteurs $b_i = \partial b/\partial v_i$, à coefficients de type `double`. La dérivation de $\mathbf{Ax} = \mathbf{b}$ par rapport à la variable v_i, $i \in [\![1, n]\!]$, donne :

$$\frac{\partial A}{\partial v_i}x + A\frac{\partial x}{\partial v_i} = \frac{\partial b}{\partial v_i}$$

Ainsi, on commencera par résoudre le système $Ax = b$. Puis, on calculera les vecteurs dérivés x_i de la solution par résolution des systèmes linéaires :

$$Ax_i = b_i - A_i x$$

Et par chance, la matrice des $n + 1$ systèmes à résoudre est toujours la matrice A. Une unique factorisation sera donc nécessaire. On voit donc qu'il est inutile de se lancer dans la modification de UMFPACK pour la rendre compatible avec la différentiation automatique, travail complexe à l'issue incertaine, puisque le résultat de l'appel aux routines de la bibliothèque peut facilement être différentié à la main.

Il faut remarquer que dans ce cas-là, les dérivées calculées coïncident avec ce que l'on aurait obtenu en appliquant la différentiation automatique au code de la bibliothèque, car c'est un solveur direct, qui résout le système à la précision machine près. Pour les solveurs itératifs, il est important de savoir qu'en calculant les dérivées des solutions aux systèmes linéaires par la méthode ci-dessus, on n'obtient pas rigoureusement les dérivées des solutions calculées par ces solveurs.

Contenu de FreeFem++-ad

Nous avons créé FreeFem++-ad à partir de la version 3.7 de FreeFem++ qui date de décembre 2010. Elle contenait donc toutes les fonctionnalités que le logiciel incluait

à cette époque, à l'exception de la version MPI et de l'affichage avec `ffglut`, que nous ne projetions de rendre compatibles avec la différentiation automatique qu'une fois le code stabilisé. Le type numérique de base pour la différentiation automatique dans les scripts de FreeFem++-ad a reçu le mot clé `treal`. Les fonctionnalités dont le mot clé a du faire l'objet d'une duplication avec changement de nom ont aussi reçu la lettre `t` en préfixe. Cela a été nécessaire pour une petite partie des identifiants globaux du logiciel, afin d'éviter d'insolubles ambiguïtés. Le tableau 4.1 dresse la liste de ces nouveaux mots clés, en précisant lesquels d'entre eux affectent négativement la stabilité à l'exécution. Ceux accompagnés d'un point d'interrogation n'ont pas été testés individuellement. Mais, quoi qu'il en soit, il est superflu de les utiliser sans avoir préalablement déclaré un maillage de type `tmesh` ou `tmesh3`, dont le maniement est périlleux.

	Description	Stabilité
treal **tcomplex**	Types numériques pour la différentiation automatique. Utilisables comme paramètres pour `fespace` (ex. : `fespace<treal>`) ou pour définir un tableau (ex. : `treal[int]`).	stable
tmesh **tmesh3**	Maillages 2D/3D avec coordonnées des points de type `treal`.	instable
tfespace	Espaces éléments finis à construire sur un maillage de type `tmesh` ou `tmesh3`.	
tmatrix **tCmatrix**	Matrices à coefficients de type `treal` ou `tcomplex`.	stable
tsolve **tproblem**	Déclaration, définition et résolution de problèmes posés sur des maillages de type `tmesh` ou `tmesh3`.	instable
tx, **ty**, **tz** **label**	Coordonnées et étiquette du point courant pour les maillages de type `tmesh` ou ou `tmesh3`.	?
tN	Normale extérieure à la frontière des maillages différentiés.	?
tP0, **tP1**, *etc.*	Identifiant de type d'éléments finis pour définir les espaces de fonctions éléments finis sur des maillages de type `tmesh` ou `tmesh3`.	?
tplot	Affichage d'une fonction éléments finis de type `tfespace.`	instable
NDiffUsed	Variable globale pour modifier le nombre de dérivées calculées.	stable
SetDiff **GetDiff** **SetVal**	Accesseurs et spécificateurs de données.	stable

Table 4.1: Nouveaux mots clés introduits dans FreeFem++-ad

On remarque dans le tableau 4.1 que c'est essentiellement la différentiation par rapport aux coordonnées des points des maillages qui rend le programme instable. En pratique, la plupart des essais menés avec FreeFem++-ad sans utiliser ces types de maillages ont conduit à des résultats corrects. On peut donc tout à fait utiliser cette version de FreeFem++ avec différentiation automatique, tant que l'on n'a pas besoin de différentier par rapport à des points du maillage. Les sections suivantes comportent quelques exemples de scripts fonctionnant tout à fait correctement.

La nature de l'instabilité reste mystérieuse. Durant le développement, nous avons pu observer des réactions inattendues de la part du compilateur `gcc`, que l'on pourrait certainement qualifier de bogues. Une utilisation trop intensive des templates (conjuguée à une programmation quelque peu anarchique) a abouti à une situation qui n'avait sans doute jamais été testée et dans laquelle le compilateur adoptait un comportement problématique. Pour développer un outil robuste de différentiation automatique, nous pensons

qu'il faudrait reprendre ce travail sous un angle d'approche tout à fait différent. Mais à l'époque où ces travaux ont été mis en œuvre, nous ne bénéficiions pas du recul et de l'expérience qui auraient pu éviter cette impasse.

4.3.3 Quelques exemples d'utilisation de FreeFem++-ad

Un problème de Poisson

Soit $\Omega =]0,1[\times]0,1[$, de frontière $\partial\Omega = \Gamma_d \cup \Gamma_n$ où $\Gamma_n = [0,1]\times\{1\}$. On se donne trois fonctions : $f \in L^2(\Omega)$, $g_d \in L^2(\Gamma_d)$ et $g_n \in L^2(\Gamma_n)$, et on définit les espaces :

$$V = \left\{ v \in H^1(\Omega) \;\middle|\; v|_{\Gamma_d} - g_d = 0 \right\} \quad \text{et} \quad V_0 = \left\{ v \in H^1(\Omega) \;\middle|\; v|_{\Gamma_d} = 0 \right\}$$

A tout $s = (s_1, s_2, s_3) \in \mathbb{R}^3$ on associe la solution de :

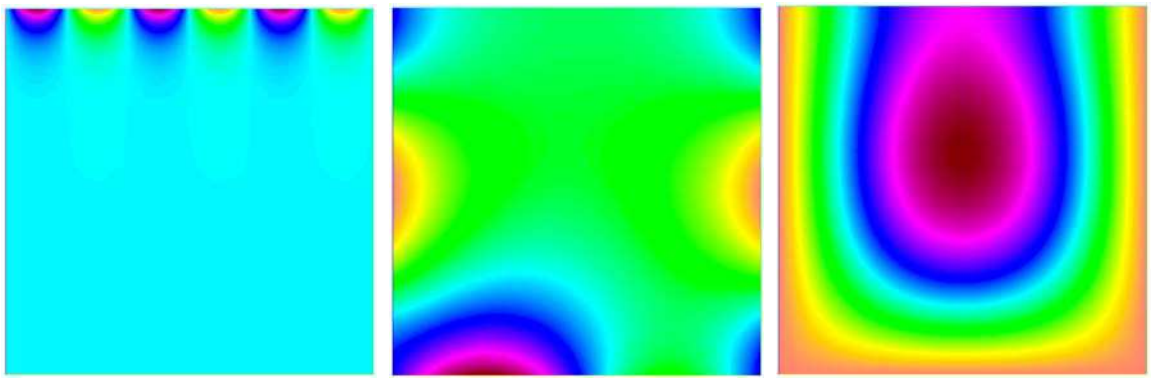
$$\begin{cases} \Delta u = s_1 f & dans\ \Omega \\ u = s_2 g_d & sur\ \Gamma_d \\ \dfrac{\partial u}{\partial n} = s_3 g_n & sur\ \Gamma_n \end{cases} \tag{4.24}$$

dont la formulation variationnelle consiste à déterminer la fonction $u \in V$ qui vérifie :

$$\forall v \in V_0\ ,\ \int_\Omega \nabla u.\nabla v - \int_\Omega s_1 f v - \int_{\Gamma_n} s_3 g_n v = 0 \tag{4.25}$$

On note $u_i = \partial u/\partial s_i$. La correspondance entre s et u est affine et on s'assure aisément que ces dérivées vérifient les formulations faibles des équations suivantes :

$$\begin{cases} \Delta u_1 = f & dans\ \Omega \\ u_1 = 0 & sur\ \Gamma_d \\ \dfrac{\partial u_1}{\partial n} = 0 & sur\ \Gamma_n \end{cases} \quad \begin{cases} \Delta u_2 = 0 & dans\ \Omega \\ u_2 = g_d & sur\ \Gamma_d \\ \dfrac{\partial u_2}{\partial n} = 0 & sur\ \Gamma_n \end{cases} \quad \begin{cases} \Delta u_3 = 0 & dans\ \Omega \\ u_3 = 0 & sur\ \Gamma_d \\ \dfrac{\partial u_3}{\partial n} = g_n & sur\ \Gamma_n \end{cases} \tag{4.26}$$



(a) $\partial u/\partial s_1$ **(b)** $\partial u/\partial s_2$ **(c)** $\partial u/\partial s_3$

FIGURE 4.4: Dérivées de la solution à l'équation 4.24 calculées par différentiation automatique.

Commentons le code FreeFem++-ad permettant de calculer ces dérivées en différentiation automatique, puis par la voie classique pour comparer les résultats. Comme de coutume dans FreeFem++, on commence par définir un maillage et un espace éléments finis :

```
mesh Th = square(50,50);
fespace Vh(Th,P1);
func f = 16.*x*(1-x)*y*(1-y);
func gn = sin(6*pi*x); //les fonctions f, g_n et g_d que l'on a choisies
func gd = sin(2*pi*x) + cos(2*pi*y);
```

On définit ensuite les variables de différentiation. La fonction `SetDiff` est utilisée pour spécifier ces variables.

```
verbosity=1;
NDiffUsed = 3;
treal[int] S=[10.,1.,2.];
for(int i=0;i<NDiffUsed;++i) SetDiff(S[i],i);
```

La résolution se poursuit de façon tout à fait habituelle. Il faut cependant déclarer l'inconnue et la fonction test comme fonctions éléments finis à composantes de type `treal`.

```
Vh<treal> u,v;
solve Poisson(u,v,solver=UMFPACK) =
    int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
  - int2d(Th)(S[0]*f*v)
  - int1d(Th,3)(S[2]*gn*v)
  + on(1,2,4,u=S[1]*gd);
```

Dans une version aboutie, la syntaxe aurait fait l'objet d'une optimisation afin d'offrir plus de confort à l'utilisation. Il aurait été également envisagé de passer directement des fonctions à composantes `treal` à la fonction d'affichage `plot`.

```
Vh U;
Vh[int] du(NDiffUsed);
U = GetVal(u);
for(int i=0;i<NDiffUsed;++i) du[i] = GetDiff(u,i);
plot(U,wait=1,cmm="the value");
for(int i=0;i<NDiffUsed;++i)
  plot(du[i],wait=1,cmm="Derivative w.r.t. s["+i+"]");
```

On complète le script en écrivant le code permettant de calculer ces dérivées par résolution des équations 4.26, et de comparer les résultats obtenus en calculant la distance en normes L^2 entre les deux solutions. Cette différence est exactement nulle quand on utilise un solveur direct, et de l'ordre de 10^{-15} avec une méthode itérative telle que le gradient conjugué. On peut donc en conclure que ce calcul par différentiation automatique fonctionne ici de manière satisfaisante.

Récupération d'une condition de Dirichlet en imposant une condition de type Neumann

Soit maintenant Ω le disque unité et $g_d \in L^2(\partial\Omega)$. A $g \in L^2(\partial\Omega)$ on associe $u_g \in H^1(\Omega)$, solution du problème variationnel :

$$\int_\Omega \nabla u_g.\nabla v - \int_\Omega u_g v - \int_{\partial\Omega} gv = 0 \ , \ \forall v \in H^1(\Omega) \tag{4.27}$$

et on définit alors la fonctionnelle sur $L^2(\partial\Omega)$ suivante :

$$J(g) = \frac{1}{2}\int_{\partial\Omega} |u_g - g_d|^2 \tag{4.28}$$

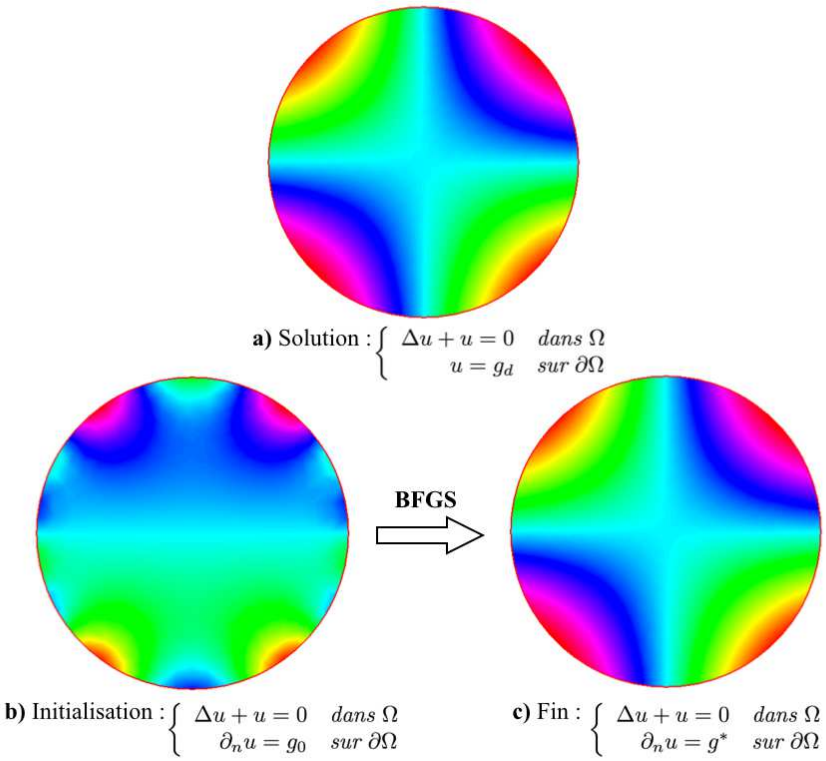


FIGURE 4.5: Détermination de la condition de Neumann g^* à imposer pour obtenir une condition de Dirichlet donnée g_d, par minimisation de la distance L^2 entre u_g et g_d sur le bord.

Par minimisation de J, on essaie de retrouver quelle condition de Neumann appliquer pour obtenir la condition de Dirichlet g_d.

Par linéarité de u_g par rapport à g, la différentielle J s'écrit :

$$\forall w \in L^2(\partial\Omega)\ ,\ dJ(g)(w) = \int_{\partial\Omega} u_w(u_g - g_d) \tag{4.29}$$

On dispose donc d'une formule exacte avec laquelle comparer les dérivées calculées par différentiation automatique.

Le code FreeFem++-ad ci-dessous propose des implémentations pour cette fonctionnelle et ses dérivées avec les deux méthodes, différentiation automatique et formule exacte, et comparaison en norme L^2 des gradients obtenus. Une minimisation de 4.28 utilisant BFGS est effectuée en fin de script.

```
1  int np=60;
2  border C(t=0,2*pi) {x=cos(t); y=sin(t); label=1;}
3
4  mesh Th = buildmesh(C(np));
5  mesh Gh = emptymesh(Th);
6  fespace Vh(Th,P1);
7  fespace Lh(Gh,P1);
8
9  func gd=x*y; //la condition de Dirichlet ciblée
10 Vh us,vs; //la solution à retrouver
11 solve Sol(us,vs)=
12     int2d(Th)(dx(us)*dx(vs)+dy(us)*dy(vs) - us*vs)
```

```
13    + on(1,us=gd);
14 plot(us,fill=1,dim=3,cmm="Solution",wait=1);
```

Les fonctions suivantes calculent la solution u_g et le gradient à l'aide de la formule 4.29. Aucune fonction de différentiation automatique n'y est utilisée. Il s'agit donc de lignes de code FreeFem++ très classiques.

```
15 func real J(real[int] &X)
16 {
17   Lh g;
18   g[]=X;
19   Vh u,v;
20   solve Prob(u,v) =
21       int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v) - u*v)
22     - int1d(Th)(g*v);
23   real res = 0.5*int1d(Th)(square(u-gd));
24   return res;
25 }
26
27 //pre-calcul
28 Vh[int] Du(Lh.ndof);
29 for(int i=0;i<Lh.ndof;++i)
30 {
31   Vh du,v;
32   Lh h=0;
33   h[][i]=1.;
34   solve dProb(du,v) =
35       int2d(Th)(dx(du)*dx(v)+dy(du)*dy(v) - du*v)
36     - int1d(Th)(h*v);
37   Du[i] = du;
38 }
39
40 func real[int] dJ(real[int] &X)
41 {
42   Lh g;
43   g[]=X;
44   Vh u,v;
45   solve Prob(u,v) =
46       int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v) - u*v)
47     - int1d(Th)(g*v);
48   real err=int2d(Th)(square(us-u));
49   plot(u,dim=3,fill=1,cmm="err="+err);
50   real[int] grad(Lh.ndof);
51   for(int i=0;i<Lh.ndof;++i)
52     grad[i] = int1d(Th)(Du[i]*(u-gd));
53   return grad;
54 }
```

Ce sont les fonctions ci-dessous qui vont calculer les dérivées par différentiation automatique. Chaque appel à la fonction `ADdJ`, qui calcule les dérivées, compare les résultats obtenus avec les deux méthodes.

```
55 NDiffUsed=Lh.ndof;
56
57 func treal tJ(real[int] &X)
58 {
59   Lh<treal> g;
60   for(int i=0;i<Lh.ndof;++i) {g[][i]=X[i]; SetDiff(g[][i],i);}
61   Vh<treal> u,v;
62   solve Prob(u,v) =
63       int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v) - u*v)
64     - int1d(Th)(g*v);
65   treal res = 0.5*int1d(Th)(square(u-gd));
66   treal err=int2d(Th)(square(us-u));
67   Vh uu=GetVal(u);
68   plot(uu,dim=3,fill=1,cmm="err="+err);
69   return res;
70 }
71
72 func real[int] ADdJ(real[int] &X)
73 {
74   treal cost = tJ(X);
75   real[int] grad(Lh.ndof);
76   for(int i=0;i<Lh.ndof;++i) grad[i] = GetDiff(cost,i);
77   real[int] diff=grad-dJ(X);
78   if(diff.l2 > 1.e-15)
79     cout << "**** Difference between AD gradient and exact one :"
80          << diff.l2 << endl;
81   return grad;
82 }
83
84 Lh start=10*cos(x)*sin(y*10);
85
86 BFGS(J,ADdJ,start[]);
```

Au cours de l'optimisation, l'avertissement de la ligne 79 n'est jamais affiché, les dérivées sont donc bien calculées de manière exacte pour ce problème. Le processus d'optimisation prend plus de temps que lorsque l'on utilise la fonction `dJ` comme implémentation du gradient pour l'appel de BFGS. Le même résultat est néanmoins obtenu en fin d'optimisation, avec une différence en norme L^2 entre la solution du problème avec la condition de Dirichlet visée et la fonction obtenue par optimisation de l'ordre de 10^{-7}. L'optimisation est donc réussie (voir figure 4.5).

Différentiation par rapport au maillage

Comme nous l'avons déjà indiqué à plusieurs reprises, les fonctionnalités qui aboutissent sur la manipulation de maillages dont les points possèdent des coordonnées de type `treal` constituent le point faible de FreeFem++-ad. Un code aussi simple que celui ci-dessous déclenche très souvent des erreurs de mémoire qui interrompent immédiatement l'exécution du programme :

```
1 tmesh Th = tsquare(10,10);//square avec différentiation automatique
2 treal a=1;
```

```
SetDiff(a,0);
Th = movemesh(Th,[a*tx,ty]);
tfespace Vh(Th,tP1);
Vh u,v;
tsolve Poisson(u,v) =
    int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v))
  - int2d(Th)(v)
  + on(1,2,3,4,u=0);
Vh du=GetDiff(u,0);
tplot(du,wait=1);
```

Il s'agit du premier test pour la validation des calculs de dérivées par rapport aux géométries. Ce problème consiste à résoudre l'équation de Poisson homogène sur un domaine dépendant d'une variable $a > 0$: $\Omega_a =]0, a[\times]0, 1[$:

$$\begin{cases} \Delta u = f & dans\ \Omega_a \\ u = 0 & sur\ \partial\Omega_a \end{cases} \tag{4.30}$$

On sait d'après [50] que la dérivée de la solution à cette équation par rapport à a est la solution u' de :

$$\begin{cases} \Delta u' = 0 & dans\ \Omega_a \\ u' = 0 & sur\ \Gamma_0 \\ u' = 1 & sur\ \Gamma_1 \end{cases}$$

où Γ_1 est le bord $\{a\}\times]0, 1[$, et $\Gamma_0 = \partial\Omega\backslash\Gamma_1$. Le but était donc de vérifier que la dérivée calculée en différentiation automatique correspondait bien à cette fonction, et c'était effectivement le cas, mais FreeFem++-ad était trop instable et il n'était pas raisonnable de poursuivre d'autres tests de validation sans avoir corrigé ce bogue.

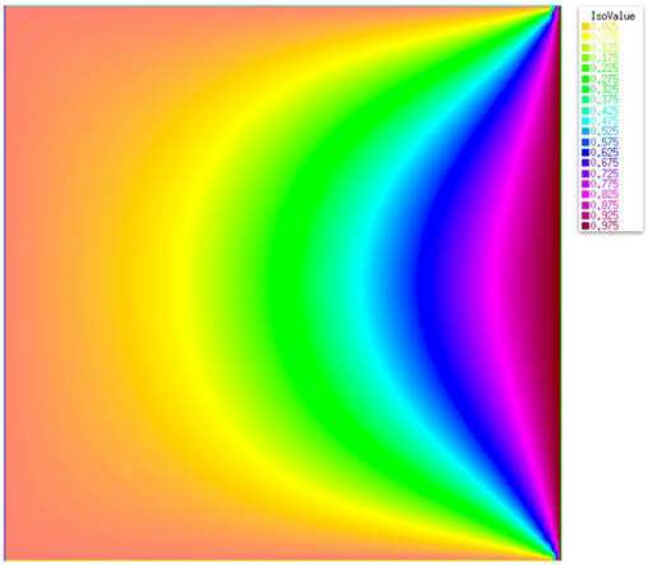


Figure 4.6: Dérivée par rapport à a de la solution du problème de Poisson 4.30 avec bord variable.

4.3.4 Supports moyens pour des cas génériques

Dans les sections théoriques précédentes, nous avons mentionné la possibilité de réduire la complexité additionnelle générée par le mode direct en limitant le stockage et le calcul

aux dérivées dont l'indice apparaît dans le support étendu 4.21. Il s'agit là de méthodes avancées dont le développement ne saurait être envisagé avant de bénéficier d'une première version basique des outils de différentiation. Par de rapides modifications des classes de différentiation, on peut néanmoins facilement concevoir un outil de calcul des supports moyens présentés en 4.22 et ainsi en obtenir les valeurs pour divers problèmes typiques de ceux qu'un utilisateur pourrait être amené à traiter à l'aide de FreeFem++. On dispose alors de plus d'éléments pour décider si un stockage en vecteurs creux sera pertinent.

Problèmes types

La question de l'utilisation d'un stockage creux des dérivées et, par extension, la détermination des supports moyens, n'ont de sens que dans le cadre de la différentiation de fonctions avec un nombre assez élevé de variables. Dans le contexte de FreeFem++, les situations auxquelles un utilisateur pourrait être confronté sont *a priori* sans limite, mais on peut cependant dresser une courte liste de problèmes types basés sur l'exploitation des éléments finis et susceptibles d'aboutir sur le calcul d'une fonctionnelle avec un grand nombre de variables. Ce sera typiquement le cas lorsque ces variables sont constituées par les composantes d'une fonction éléments finis. Il s'agit par exemple de calculs comportant une étape de résolution d'un problème aux limites, par la méthode des éléments finis, que l'on souhaiterait dériver par rapport aux composantes de la fonction intervenant dans un éventuel second membre, ou jouant le rôle de coefficients au sein de la forme bilinéaire, ou bien encore celui de condition aux bords, *etc.*

Nous allons donc nous intéresser aux supports moyens, médians et maximaux en fonction du nombre de variables pour chacun des problèmes types suivants, dans lesquels Ω désigne un ouvert de $\mathbb{R}^2$, $\mathcal{T}_h$ une triangulation de Ω et V_h l'espace des fonctions P^1 sur chacun des éléments de $\mathcal{T}_h$, ainsi que V_h^0 le sous-espace des éléments V_h s'annulant sur le bord. L'ensemble G_h désignera quant à lui l'espace des fonctions P^1 définies sur le bord de la triangulation. On notera respectivement N_V et N_G les dimensions des espaces V_h et G_h.

Problème 1 : On définit l'application F_1 de V_h dans V_h qui à toute fonction $\kappa \in V_h$ associe la fonction $u \in V_h$ vérifiant :

$$\forall v \in V_h,\ \int_\Omega \kappa \nabla u.\nabla v - \int_\Omega uv - \int_\Omega fv = 0$$

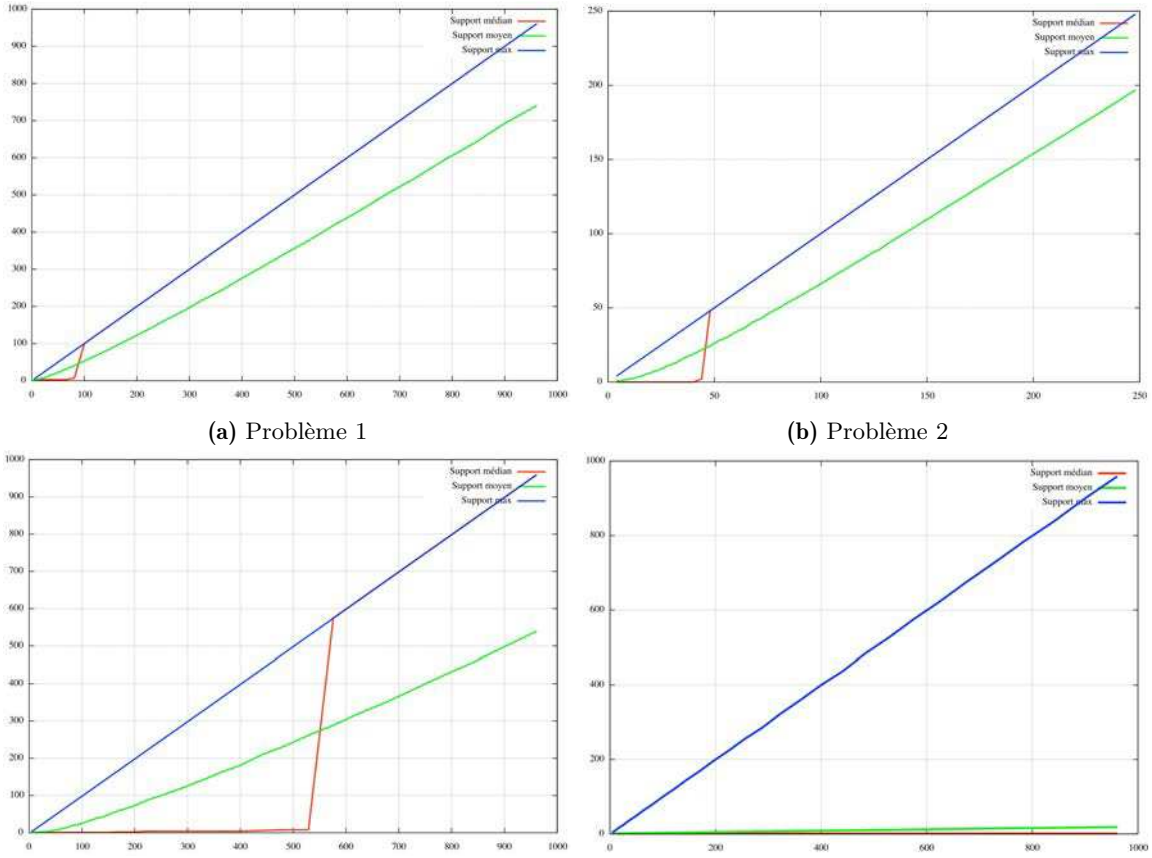
où f est une fonction de $L^2(\Omega)$.

Problème 2 : L'application $F_2 : G_h \to V_h$ associe à tout $g \in G_h$, $u \in V_h$ vérifiant :

$$\forall v \in V_h,\ \int_\Omega \nabla u.\nabla v - \int_\Omega uv - \int_\Omega fv - \int_{\partial\Omega} gv = 0$$

Problème 3 : L'application $F_3 : V_h \to V_h$ associe à tout $f \in V_h$, $u \in V_h$ vérifiant :

$$\forall v \in V_h,\ \int_\Omega \nabla u.\nabla v - \int_\Omega uv - \int_\Omega fv = 0$$



(a) Problème 1 **(b)** Problème 2

(c) Problème 3 **(d)** Problème 4

Figure 4.7: Supports maximaux, moyens et médians dans la résolution de problèmes types rencontrés en éléments finis avec différentiation automatique.

Problème 4 : $S : V_h \to \mathbb{R}$, fonctionnelle de surface du graphe d'un élément de V_h :

$$S(u) = \int_\Omega \sqrt{1 + |\nabla u|^2}$$

Les résultats des calculs de supports des figures 4.7 indiquent clairement que, lors de la résolution d'un problème aux limites par éléments finis avec différentiation automatique, une fraction importante des dérivées sont non nulles. Les supports moyens calculés pour ces types de problèmes sont relativement élevés et les supports médians ne diffèrent du nombre de variables de dérivation que lorsque le domaine de calcul a été maillé grossièrement. On en déduit que pour ces problèmes, l'utilisation de vecteurs creux pour le calcul des dérivées par différentiation automatique ne réduira que peu la complexité, et serait même susceptible d'aggraver celle-ci.

En revanche, pour l'intégration plus ou moins directe de la fonction éléments finis dont les composantes servent de variables de dérivation telle que la fonctionnelle d'aire qui apparaît dans les problèmes de surfaces minimales, le support moyen reste très faible et des outils de différentiation plus adaptés tels que ceux proposés par [91] pourront certainement se montrer efficaces.

4.3.5 Perspectives

Nous avons donc développé une version de FreeFem++ pourvue d'outils de différentiation automatique qui fonctionnent relativement bien tant que les scripts utilisés ne font pas appel aux fonctionnalités de différentiation par rapport à des points du maillage. Les quelques problèmes tests menés dans ce cas suggèrent que ces outils calculent ce pour quoi ils ont été conçus, et cela semble être aussi le cas pour les fonctionnalités instables, bien que ces dernières n'aient pas fait l'objet d'une validation en bonne et due forme.

La syntaxe nécessiterait par ailleurs d'être repensée car celle-ci est encore trop confuse. Déterminer dans quel cas il faut utiliser la version avec préfixe `t` d'un mot clé ou sa version usuelle n'est en effet pas assez intuitif et il faudrait beaucoup de temps à un usager pour s'habituer à ce système. Il était cependant inutile d'optimiser l'interface utilisateur tant que les outils eux-mêmes n'étaient pas entièrement fonctionnels. Cette partie du travail a donc été laissée en suspens.

Se pose finalement la question de savoir si ce travail pourra, à plus ou moins long terme, aboutir sur les nouveaux outils qui avaient été envisagés dans le projet de thèse. On pourrait imaginer retirer les modifications introduites pour la dérivation par rapport à des éléments de la géométrie et ne conserver que les aménagements qui fonctionnent raisonnablement bien. Il faut cependant savoir que ce prototype ayant été conçu à partir d'une version de FreeFem++ aujourd'hui vieille de quatre ans, les efforts à déployer pour sa mise à jour seront considérables. Une autre approche serait, à l'instar de la plupart des nouvelles fonctionnalités qui ont été dernièrement introduites dans le logiciel, de proposer les outils de différentiation automatique par l'intermédiaire d'un module externe. On éviterait ainsi d'altérer le code du logiciel avec toutes les répercussions que de telles modifications du noyau entraînent en termes de développement. En contrepartie, il faudrait dans ce cas repartir à une phase très primitive du développement, ce qui impliquerait nécessairement un réengagement conséquent, même si l'expérience préalablement accumulée avec ce premier jet faciliterait la tâche. De plus, Il n'est pas certain, et ce pour des raisons techniques, qu'il soit possible de remplir les exigences du cahier des charges de cette manière. Dans tous les cas, un réinvestissement assez lourd sera nécessaire.

Enfin, une piste intéressante que nous avions évoquée pendant la thèse, et dont nous faisons mention dans ce manuscrit, est celle consistant en l'extension des arguments des formulations variationnelles acceptées par FreeFem++ à des combinaisons non nécessairement bilinéaires ou linéaires des fonctions éléments finis intervenant dans la définition des problèmes. On pourrait alors, en se basant sur le paradigme de la différentiation symbolique, développer des outils qui dériveraient automatiquement problèmes adjoints et tangents. En plus de permettre la dérivation de fonctionnelles, cela peut également être utile pour l'automatisation de la résolution de certains problèmes, telles les formulations variationnelles non linéaires que l'on pourrait traiter par défaut avec une méthode de Newton automatisée. Même s'il n'est pas certain qu'une telle systématisation du traitement des problèmes non linéaires se révèle toujours pertinente, les questions qu'ils suscitent ne trouvant le plus souvent de réponse qu'au cas par cas, le sujet mériterait toutefois d'être creusé.

Annexe A

Les algorithmes d'optimisation de FreeFem++

A.1 Gradient conjugué non linéaire

Rappelons avant tout l'algorithme du gradient conjugué standard qui permet de minimiser une fonction de la forme $f(x) = \frac{1}{2}(Ax, x) - (b, x)$ ou, de manière équivalente, de résoudre le système linéaire $Ax = b$ (remarquons à ce propos que cela équivaut à $g(x) = \nabla f(x) = 0$) :

Algorithm 4 Gradient Conjugué Non-Linéaire

$d_0 = r_0 = -g(x_0)$
while $\|r_k\| > \epsilon$ **do**
$d_{k+1} = r_k - \sum_{i=0}^{k} \frac{(Ad_i, r_k)}{(Ad_i, d_i)} d_i$
$\alpha_{k+1} = \frac{(d_{k+1}, r_k)}{(Ad_{k+1}, d_{k+1})}$
$x_{k+1} = x_k + \alpha_{k+1} d_{k+1}$
$r_{k+1} = -g(x_{k+1})$
end while

A.2 BFGS

La méthode BFGS (Broyden, Fletcher, Goldfarb et Shanno) est une méthode de type quasi-Newton. Il s'agit donc d'une méthode de Newton dans laquelle la matrice hessienne, ou plus exactement dans le cas de cette méthode, son inverse, est approchée plutôt que directement calculée, ce qui permet de pouvoir l'appliquer à des fonctions pour lesquelles on ne dispose pas d'une implémentation des dérivées secondes, tout en conservant des propriétés de convergence proches de celles de la méthode de Newton. Elle s'applique à la résolution de problèmes sans contraintes :

$$\textit{trouver } x^* \textit{ tel que } x^* = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} f(x) \tag{A.1}$$

L'écriture de l'algorithme ne requiert que la régularité C^1 pour la fonction coût $f : \mathbb{R}^n \to \mathbb{R}$. Les preuves de convergence locale nécessitent en revanche une fonction coût deux fois

dérivable avec une hessienne $x \mapsto \nabla^2 f(x)$ au moins lipschitzienne (voir [16]). Il faut donc s'attendre à des comportements erratiques lorsque l'algorithme est utilisé sur des fonctions qui ne sont pas assez lisses.

Algorithm 5 - BFGS

Initialisations :

- $W_0 := I_n$: initialisation de l'approximation de l'inverse de $\nabla^2 f$
- x_0 : point initial
- Nombre maximum d'itérations N, tolérance ϵ
- $k \leftarrow 0$: initialisation du compteur

while $\|\nabla f(x_k)\| > \epsilon$ **do**

if $k > N$ **then**

Arrêter l'algorithme qui n'a pas réussi à converger

end if

Calculer la direction de descente : $d_k \leftarrow W_k \nabla f(x_K)$

Déterminer α_k tel que $\alpha_k = \underset{\alpha \in \mathbb{R}}{\operatorname{argmin}} f(x_k + \alpha d_k)$, par une recherche linéaire

$x_{k+1} \leftarrow x_k + \alpha_k d_k$

$s_k \leftarrow \alpha_k d_k = x_{k+1} - x_k$

$y_k \leftarrow \nabla f(x_{k+1}) - \nabla f(x_k)$

Mise à jour de l'approximation de $\nabla^2 f^{-1}$:

$$W_{k+1} \leftarrow W_k - \frac{s_k {y_k}^{\mathrm{T}} W_k + W_k y_k {s_k}^{\mathrm{T}}}{(y_k, s_k)} + \left[1 + \frac{(y_k, W_k y_k)}{(y_k, s_k)}\right] \frac{s_k {s_k}^{\mathrm{T}}}{(y_k, s_k)}$$

end while

La recherche linéaire utilisée dans l'implémentation [64] incluse dans FreeFem++ s'inspire d'un algorithme proposé dans [37], chapitre 6, dans lequel les applications $\alpha \mapsto f(x + \alpha d)$ sont localement approchées par des polynômes de degré trois en α.

A.3 IPOPT

Nous donnons ici le détail étape par étape de l'algorithme IPOPT tel que présenté dans [92]. On suppose que f est une fonction C^2 de $\mathbb{R}^n$ dans $\mathbb{R}$, c une fonction C^2 de $\mathbb{R}^n$ dans $\mathbb{R}^m$, et x^l et x^u deux éléments fixés de $\mathbb{R}^n$ tels que $x^l \leq x^u$ (inégalité de composante à composante de même indice). La généralisation à des vecteurs bornes de composantes dans $\overline{\mathbb{R}}$ pourra être trouvée dans [92].

$$\begin{aligned} &\textit{trouver :} \quad x^* = \underset{x \in V}{\operatorname{argmin}}\ f(x) \\ &\textit{avec} \quad V = \left\{ x \in \mathbb{R}^n \mid c(x) = 0 \quad \text{et} \quad x^l \leq x \leq x^u \right\} \end{aligned} \tag{A.2}$$

On reprendra quelques-unes des notations de [92]. Nous noterons par exemple la mesure de violation des contraintes d'égalité $\theta(x) = \|c(x)\|$ afin d'assurer la cohérence des notations.

L'erreur pour le test de convergence globale est celle définie en 2.21 :

$$E_\mu(x,\lambda,z_u,z_l) = \max\left\{\frac{\left\|\nabla f(x) + J_c(x)^{\mathrm{T}}\lambda + z_u - z_l\right\|_\infty}{s_d}, \|c(x)\|_\infty, \right. \\ \left. \frac{\|(X^u - X)z_u - \mu e\|_\infty}{s_c}, \frac{\left\|(X - X^l)z_l - \mu e\right\|_\infty}{s_c}\right\}$$

On appelle $\mathcal{B}_\mu$ le sous-problème de minimisation de la fonction barrière de paramètre μ :

$$B(x,\mu) = f(x) - \mu\sum_{i=1}^{n}\ln(x_i - x_i^l) - \mu\sum_{i=1}^{n}\ln(x_i^u - x_i)$$

sous la contrainte $c(x) = 0$. On fera également référence au système 2.23 après correction d'inertie par $\mathcal{S}_\mu^{\delta_w,\delta_c}(x,\lambda,z^l,z^u)$:

$$\begin{pmatrix}\nabla^2\mathcal{L}(x,\lambda) + \Sigma(x,z^l,z^u) + \delta_w I_n & J_c(x)^{\mathrm{T}} \\ J_c(x) & -\delta_c I_m\end{pmatrix}\begin{pmatrix}d_x \\ d_\lambda\end{pmatrix} = -\begin{pmatrix}\nabla B(x,\mu) + J_c(x)^{\mathrm{T}}\lambda \\ c(x)\end{pmatrix} \quad \text{(A.3)}$$

dans lequel $\Sigma(x,z^l,z^u) = (X_u - X)^{-1}Z_u + (X - X_l)^{-1}Z_l$ est une matrice carrée de taille n.

Les valeurs des constantes apparaissant dans l'algorithme utilisé pour l'implémentation IPOPT pourront être retrouvées encore une fois dans [92].

Détails de l'algorithme IPOPT :

0) Données et initialisations

- point de départ : $(x_0,\lambda_0,z_0^l,z_0^u) \in \mathbb{R}^n\times\mathbb{R}^m\times\mathbb{R}^n\times\mathbb{R}^n$
- paramètre de barrière initial : $\mu_0 > 0$
- diverses constantes : $\epsilon_{\mathrm{tol}} > 0$, $s_{\max} \geq 1$, $\kappa_\epsilon > 0$, $\kappa_\mu \in]0,1[$, $\theta_\mu \in]1,2[$, $\tau_{\min} \in]0,1[$, $\kappa_\Sigma > 1$, $\theta_{\max} \in]\theta(x_0),+\infty]$, $\theta_{\min} > 0$, γ_θ, $\gamma_\varphi \in]0,1[$, $\delta > 0$, $\gamma_\alpha \in]0,1]$, $s_\theta > 1$, $s_\varphi \geq 1$, $\eta_\varphi \in]0,\frac{1}{2}[$, $\kappa_{\mathrm{soc}} \in]0,1[$, $p^{\max} \in \{0,1,2,\ldots\}$
- constantes correcteur d'inertie : $0 < \bar{\delta}_w^{\min} < \bar{\delta}_w^{\max}$, $\bar{\delta}_c > 0$, $0 < \kappa_w^- < 1 < \kappa_w^+ < \bar{\kappa}_w^+$, $\kappa_c \geq 0$.
- initialisation des compteurs : $l \leftarrow 0$, $k \leftarrow 0$
- initialisation du filtre : $\mathcal{F}_0 \leftarrow \{(\theta,\varphi) \in \mathbb{R}^2 \mid \theta \geq \theta_{\max}\}$
- initialisation : $\tau_0 \leftarrow \max\{\tau_{\min}, 1-\mu_0\}$, $\delta_w^{\mathrm{last}} \leftarrow 0$

1) Test de convergence globale - si $E_0(x_k,\lambda_k,z_k^l,z_k^u) \leq \epsilon_{\mathrm{tol}}$ (équation 2.21, rappelée ci-dessus, avec $\mu = 0$) alors l'algorithme prend fin en ayant convergé : **STOP [CONVERGENCE]**, sinon continuer en 2).

2) Test de convergence pour le sous-problème $\mathcal{B}_{\mu_j}$ - si $E_{\mu_j}(x_k,\lambda_k,z_k^l,z_k^u) \leq \kappa_\epsilon\mu_j$ alors

- $\mu_{j+1} \leftarrow \max\left\{\frac{\epsilon_{\mathrm{tol}}}{10}, \min\left\{\kappa_\mu\mu_j, {\mu_j}^{\theta_\mu}\right\}\right\}$, $\tau_{j+1} \leftarrow \max\{\tau_{\min}, 1-\mu_j\}$ et $j \leftarrow j+1$
- Réinitialisation du filtre : $\mathcal{F}_k \leftarrow \{(\theta,\varphi) \in \mathbb{R}^2 \mid \theta \geq \theta_{\max}\}$
- Si $k = 0$ répéter cette étape 2), sinon passer à 3).

3) Calcul des directions de descente - déterminer $(d_k^x, d_k^\lambda, d_k^l, d_k^u) \in \mathbb{R}^n\times\mathbb{R}^m\times\mathbb{R}^n\times\mathbb{R}^n$

3.1 Tenter la factorisation de la matrice de $\mathcal{S}_{\mu_j}^{0,0}(x_k,\lambda_k,z_k^l;z_k^u)$ (système sans correction d'inertie). Si la matrice est inversible et possède l'inertie $(n,m,0)$, utiliser cette factorisation pour le calcul de d_k^x et d_k^λ puis aller en 3.7, sinon continuer avec l'étape suivante 3.2.

3.2 Si la matrice en 3.1 possède une ou plusieurs valeur(s) propre(s) nulle(s) alors $\delta_c \leftarrow \bar{\delta}_c \, \mu_j{}^{\kappa_c}$ sinon $\delta_c \leftarrow 0$.

3.3 Si $\delta_w^{\text{last}} = 0$, alors $\delta_w \leftarrow \bar{\delta}_w^0$, sinon $\delta_w \leftarrow \max\left\{\bar{\delta}_w^{\min}, \kappa_w^- \delta_w^{\text{last}}\right\}$.

3.4 Tenter de factoriser la matrice du système $\mathcal{S}_{\mu_j}^{\delta_w,\delta_c}(x_k, \lambda_k, z_k^l; z_k^u)$. Si l'inertie est $(n, m, 0)$ alors $\delta_w^{\text{last}} \leftarrow \delta_w$, calculer d_k^x et d_k^λ avec cette factorisation et aller en 3.7, sinon continuer en 3.5.

3.5 Si $\delta_w^{\text{last}} = 0$, alors $\delta_w \leftarrow \bar{\kappa}_w^+ \delta_w$, sinon $\delta_w \leftarrow \kappa_w^+ \delta_w$.

3.6 Si $\delta_w > \bar{\delta}_w^{\max}$, interrompre la recherche des directions de descente et aller directement en 8) pour la phase de restauration, sinon retourner à l'étape 3.4.

3.7 Calculer les directions associées à z_k^l et z_k^u selon les formules ci-dessous puis aller à l'étape 4) :

$$d_k^l \leftarrow -(X_k - X_k^l)^{-1} Z_k^l d_k^x + z_k^l - (X_k - X_k^l)^{-1} \mu e$$
$$d_k^u \leftarrow \quad (X_k^u - X_k)^{-1} Z_k^u d_k^x + z_k^u - (X_k^u - X_k)^{-1} \mu e$$

4) Recherche linéaire

4.1 Initialisations :

$$\alpha_k^{\max} \leftarrow \max\{\alpha \in]0,1] \mid x_k + \alpha d_k^x \geq (1-\tau_j) x_k\}$$
$$\alpha_{k,0} \leftarrow \alpha_k^{\max} \text{ et } l \leftarrow 0.$$

4.2 Proposer un nouveau point : $x_{k,l} := x_k + \alpha_{k,l} d_k^x$

4.3 Tester l'acceptabilité vis-à-vis du filtre : si $(\theta(x_{k,l}), B(x_{k,l}, \mu_j)) \in \mathcal{F}_k$, le pas $\alpha_{k,l}$ est rejeté et on poursuivra par l'étape 4.5.

4.4 Analyse de la réduction de la fonction coût et de la violation des contraintes pour le nouveau point :

- *Cas 1 - si* $\theta(x_k) \leq \theta_{\min}$,$\nabla B(x_k, \mu_j).d_k^x < 0$ et $\alpha_{k,l}\,(-\nabla B(x_k, \mu_j).d_k^x)^{s_\varphi} > \delta\, \theta(x_k)^{s_\theta}$: si $B(x_{k,l}, \mu_j) \leq B(x_k, \mu_j) + \eta_\varphi \alpha_{k,l} \nabla B(x_k, \mu_j).d_k^x$ alors $x_{k+1} \leftarrow x_{k,l}$ et aller en 5), sinon aller en 4.5.
- *Cas 2 - si la condition du cas 1 n'est pas vérifiée :* si $\theta(x_{k,l}) \leq (1-\gamma_\theta)\theta(x_k)$ ou $B(x_{k,l}, \mu_j) \leq B(x_k, \mu_j) - \gamma_\varphi \theta(x_k)$ alors $x_{k+1} \leftarrow x_{k,l}$ et aller en 5), sinon aller en 4.5.

4.5 Correction au second ordre (SOC) - Si $l > 0$ ou $\theta(x_{k,0}) < \theta(x_k)$ cette phase n'est pas effectuée et on passe directement à l'étape 4.10. Sinon, on initialise le compteur $p \leftarrow 1$, $\theta_{\text{old}}^{\text{soc}} \leftarrow \theta(x_k)$ et $c_k^{\text{soc}} \leftarrow \alpha_{k,0} c(x_k) + c(x_k + \alpha_{k,0} d_k^x)$.

4.6 Calculer :

- $d_k^{x,\text{cor}}$ et d_k^λ en résolvant le système linéaire $\mathcal{S}_{\mu_j}^{\delta_w,\delta_c}(x_k, \lambda_k, z_k^l; z_k^u)$ avec le second membre $-\begin{pmatrix} \nabla B(x_k, \mu_j) + J_c(x_k)^{\mathrm{T}} \lambda_k \\ c_k^{\text{soc}} \end{pmatrix}$, dont la matrice a déjà fait l'objet d'une factorisation en 3.4.
- $\alpha_k^{\text{soc}} \leftarrow \max\{\alpha \in]0,1] \mid x_k + \alpha d_k^{x,\text{cor}} \geq (1-\tau_j) x_k\}$
- $x_k^{\text{soc}} \leftarrow x_k + \alpha_k^{\text{soc}} d_k^{x,\text{cor}}$

4.7 Test d'adéquation au filtre : si $(\theta(x_k^{\text{soc}}), B(x_k^{\text{soc}}, \mu_j)) \in \mathcal{F}_k$, x_k^{soc} est rejeté et on passe à l'étape 4.10.

4.8 Analyse de la réduction de la fonction coût et de la violation des contraintes pour le nouveau point :

- *Cas 1 - si* $\theta(x_k) \leq \theta_{\min}$,$\nabla B(x_k, \mu_j).d_k^x < 0$ et $\alpha_{k,0}\,(-\nabla B(x_k, \mu_j).d_k^x)^{s_\varphi} > \delta\, \theta(x_k)^{s_\theta}$: si $B(x_k^{\text{soc}}, \mu_j) \leq B(x_k, \mu_j) + \eta_\varphi \alpha_{k,0} \nabla B(x_k, \mu_j).d_k^x$ alors $x_{k+1} \leftarrow x_k^{\text{soc}}$ et aller en 5), sinon aller en 4.9.
- *Cas 2 - si la condition du cas 1 n'est pas vérifiée :* si $\theta(x_k^{\text{soc}}) \leq (1-\gamma_\theta)\theta(x_k)$ ou $B(x_k^{\text{soc}}, \mu_j) \leq B(x_k, \mu_j) - \gamma_\varphi \theta(x_k)$ alors $x_{k+1} \leftarrow x_k^{\text{soc}}$ et aller en 5), sinon aller en 4.9.

4.9 Deuxième correction au second ordre : si $p = p^{\max}$ ou $\theta(x_k^{\text{soc}}) > \kappa_{\text{soc}}\theta_{\text{old}}^{\text{soc}}$, aller en 4.10, sinon, $p \leftarrow p+1$, $c_k^{\text{soc}} \leftarrow \alpha_k^{\text{soc}} c_k^{\text{soc}} + c(x_k^{\text{soc}})$ et $\theta_{\text{old}}^{\text{soc}} \leftarrow \theta(x_k^{\text{soc}})$, puis retour à 4.6.

4.10 Proposition d'un nouveau pas

- $\alpha_{k,l+1} \leftarrow \frac{1}{2}\alpha_{k,l}$ et $l \leftarrow l+1$
- $$\alpha_k^{\min} := \gamma_\alpha \cdot \begin{cases} \min\left\{\gamma_\theta, \dfrac{\gamma_\varphi \theta(x_k)}{-\nabla B(x_k,\mu_j).d_k^x}, \dfrac{\delta\,(\theta(x_k))^{s_\theta}}{(-\nabla B(x_k,\mu_j).d_k^x)^{s_\varphi}}\right\} \\ \qquad\qquad \text{si } \nabla B(x_k,\mu_j).d_k^x < 0 \text{ et } \theta(x_k) \le \theta^{\min} \\ \min\left\{\gamma_\theta, \dfrac{\gamma_\varphi \theta(x_k)}{-\nabla B(x_k,\mu_j).d_k^x}\right\} \\ \qquad\qquad \text{si } \nabla B(x_k,\mu_j).d_k^x < 0 \text{ et } \theta(x_k) > \theta^{\min} \\ \gamma_\theta \\ \qquad\qquad \text{dans les autres cas} \end{cases}$$
- si $\alpha_{k,l} < \alpha_k^{\min}$, aller à l'étape 8), sinon, retour à 4.2.

5) Le pas $\alpha_{k,l}$ est accepté : si il y a eu correction du second ordre (si l'étape précédente était 4.8) $\alpha_k \leftarrow \alpha_k^{\text{soc}}$, sinon, $\alpha_k \leftarrow \alpha_{k,l}$, puis :

- calculer :
  $\alpha_k^l := \max\left\{\alpha \in]0,1] \mid z_k^l + \alpha d_k^l \ge (1-\tau_j) z_k^l\right\}$
  $\alpha_k^u := \max\left\{\alpha \in]0,1] \mid z_k^u + \alpha d_k^u \ge (1-\tau_j) z_k^u\right\}$
- mettre à jour les multiplicateurs de Lagrange :
  $\lambda_{k+1} \leftarrow \lambda_k + \alpha_k d_k^\lambda$
  $z_{k+1}^l \leftarrow z_k^l + \alpha_k^l d_k^l$
  $z_{k+1}^u \leftarrow z_k^u + \alpha_k^u d_k^u$
- correction des multiplicateurs associés aux bornes - Pour tout $i \in [\![1,n]\!]$:
  $$z_{k+1}^{l\,(i)} \leftarrow \max\left\{\min\left\{z_{k+1}^{l\,(i)}, \frac{\kappa_\Sigma \mu_j}{x_{k+1}^{(i)}}\right\}, \frac{\mu_j}{\kappa_\Sigma x_{k+1}^{(i)}}\right\}$$
  $$z_{k+1}^{u\,(i)} \leftarrow \max\left\{\min\left\{z_{k+1}^{u\,(i)}, \frac{\kappa_\Sigma \mu_j}{x_{k+1}^{(i)}}\right\}, \frac{\mu_j}{\kappa_\Sigma x_{k+1}^{(i)}}\right\}$$

6) Mise à jour du filtre - Si l'une des conditions suivantes n'est pas remplie :

- $\nabla B(x_k,\mu_j).d_k^x < 0$
- $\alpha_k\,(-\nabla B(x_k,\mu_j).d_k^x)^{s_\varphi} > \delta\,(\theta(x_k))^{s_\theta}$
- $B(x_{k+1},\mu_j) \le B(x_k,\mu_j) + \eta_\varphi \alpha_k \nabla B(x_k,\mu_j).d_k^x$

alors $\mathcal{F}_{k+1} \leftarrow \mathcal{F}_k \cup \left\{(\theta,\varphi) \in \mathbb{R}^2 \mid \theta \ge (1-\gamma_\theta)\theta(x_k) \text{ et } \varphi \ge B(x_k,\mu_j) - \gamma_\varphi \theta(x_k)\right\}$

7) Augmenter le compteur global $k \leftarrow k+1$ puis entamer l'itération suivante en retournant en 1).

8) Phase de restauration - Mettre à jour le filtre inconditionnellement :

$\mathcal{F}_{k+1} \leftarrow \mathcal{F}_k \cup \left\{(\theta,\varphi) \in \mathbb{R}^2 \mid \theta \ge (1-\gamma_\theta)\theta(x_k) \text{ et } \varphi \ge B(x_k,\mu_j) - \gamma_\varphi \theta(x_k)\right\}$

Essayer de construire une nouvelle itérée x_{k+1} en faisant décroître $\theta(x)$ de manière à ce que $(\theta(x_{k+1}), B(x_{k+1},\mu_j)) \notin \mathcal{F}_{k+1}$, puis poursuivre en 7). L'algorithme utilisé dans IPOPT pour cette tâche est décrit dans [92].

A.4 CMA-ES

Initialisations :

- Constantes : λ, μ, $w_1, w_2, \dots w_\mu$, $\mu_w = \left(\sum_{i=1}^{\mu} w_i\right)^{-1}$, $c_\sigma > 1$, $c_c > 1$, $d_\sigma \simeq 1$, $\alpha \simeq \frac{3}{2}$, $c_1 \simeq \frac{2}{n^2}$, $c_\mu \le 1 - c_1$

- $m_0 \in \mathbb{R}^n$ - la moyenne initiale (équivalente au point initial)
- σ_0 - l'écart type global initial
- $C_0 = I_n$ - matrice de covariance initiale
- $p_\sigma^0 = 0$ et $p_c^0 = 0$ - deux vecteurs de $\mathbb{R}^n$
- $k \leftarrow 0$ - compteur d'itérations

while critères d'arrêts non vérifiés **do**

Echantillonner $x_i^k, i \in [\![1, \lambda]\!]$ selon la loi $\mathcal{N}(m_k, \sigma_k C_k)$

Evaluer $f_i^k \leftarrow f(x_i^k),\ \forall i \in [\![1, \lambda]\!]$

Trier la population par rapport aux images par f : $x_i^k \leftarrow x_{s(i)}^k$, où s est une permutation de $[\![1, \lambda]\!]$ telle que $f_{s(1)}^k \leq f_{s(2)}^k \leq \cdots \leq f_{s(\lambda)}^k$.

Mises à jour :

- $m_{k+1} \leftarrow \sum_{i=1}^{\mu} w_i x_i^k$
- $p_\sigma^{k+1} \leftarrow (1 - c_\sigma) p_\sigma^k + \sqrt{1 - (1 - c_\sigma)^2}\sqrt{\mu_w} C_k^{-1/2} \dfrac{m_{k+1} - m_k}{\sigma_k}$
- $p_c^{k+1} \leftarrow (1 - c_c) p_c^k + \mathbf{1}_{[0, \alpha\sqrt{n}]}\left(\left\|p_\sigma^{k+1}\right\|\right) \sqrt{1 - (1 - c_c)^2}\sqrt{\mu_w} \dfrac{m_{k+1} - m_k}{\sigma_k}$
- $c_s^k \leftarrow \left(1 - \mathbf{1}_{[0, \alpha\sqrt{n}]}\left(\left\|p_\sigma^{k+1}\right\|\right)^2\right) c_1 c_c (2 - c_c)$
- $C_{k+1} = (1 - c_1 - c_\mu + c_s^k) C_k + c_1 p_c^{k+1} {p_c^{k+1}}^{\mathrm{T}} + c_\mu \sum_{i=1}^{\mu} w_i \dfrac{x_i^k - m_k}{\sigma_k} \left(\dfrac{x_i^k - m_k}{\sigma_k}\right)^{\mathrm{T}}$
- $\sigma_{k+1} \leftarrow \sigma_k \exp\left[\dfrac{c_\sigma}{d_\sigma}\left(\dfrac{\left\|p_\sigma^{k+1}\right\|}{E\left\|\mathcal{N}(0, I_n)\right\|} - 1\right)\right]$

end while

Renvoyer m_k ou x_1^k

Annexe B

Extraits de scripts FreeFem++ des applications

B.1 Codes pour les surfaces minimales

```
load "ff-Ipopt"
load "Element_P4dc"
load "medit"
load "msh3"
bool SB=1;
int np=40;
func theta=x!=0 ? asin(y/sqrt(x^2+y^2)) : 0.;
//func g=2*exp(-(x^2)/(2*0.1));
//func g=cos(4*theta);
//func g=log(cos(x))-log(cos(y));
func g=cos(4*theta);
int[int] labels=[1,1,1,1];

//border C(t=0,2*pi) {x=cos(t); y=sin(t); label=1;}
border C(t=0,2*pi)
{
        real r=0.5*sin(8*t) + 1.;
        x=r*cos(t);
        y=r*sin(t);
        label=1;
}
mesh Th = buildmesh(C(2*pi*np));
//real eps=0.1;
//mesh Th =
//  square(np*pi*(1.-eps),
//         np*pi*(1-eps),
//         [(x-0.5)*pi*(1.-eps),
//          (y-0.5)*pi*(1.-eps)],
//         label=labels);
//plot(Th,wait=1);
fespace Vh(Th,P1);
```

```
fespace Wh(Th,P4dc);
macro Grad(u) [dx(u),dy(u)] //EOM

int iter=0;
func real A(real[int] &X)
{
        Vh u; u[]=X;
        Wh alpha = 1./sqrt(1. + Grad(u)'*Grad(u));
        real area = int2d(Th)(sqrt(1 + Grad(u)'*Grad(u)));
        return area;
}
func real[int] dA(real[int] &X)
{
        Vh u; u[]=X;
        Wh alpha = 1./sqrt(1. + Grad(u)'*Grad(u));
        varf vfdA(w,v) = int2d(Th)(alpha * Grad(u)'*Grad(v));
        varf vfdAb(w,v) = int2d(Th)(alpha * Grad(u)'*Grad(v)) + on(1,w=0);
        real[int] g(Vh.ndof);
        if(SB) g = vfdA(0,Vh); else g = vfdAb(0,Vh);
        return g;
}
matrix hessian;
func matrix d2A(real[int] &X)
{
        Vh u; u[]=X;
        Wh alpha = 1./sqrt(1. + Grad(u)'*Grad(u)),
           beta = 1./sqrt(1. + Grad(u)'*Grad(u))^3;
        varf vfd2A(w,v) =
          int2d(Th)(  alpha * Grad(w)'*Grad(v)
                    - beta*(Grad(u)'*Grad(v))*(Grad(u)'*Grad(w)));
        varf vfd2Ab(w,v) =
            int2d(Th)(  alpha * Grad(w)'*Grad(v)
                    - beta*(Grad(u)'*Grad(v))*(Grad(u)'*Grad(w)))
          + on(1,w=0);
        if(SB) hessian = vfd2A(Vh,Vh); else hessian = vfd2A(Vh,Vh);
        plot(u,dim=3,fill=1);
        return hessian;
}

for(int i=0;i<3;++i)
{
  varf onb(w,v) = on(1,w=1);
  Vh Onb; Onb[] = onb(0,Vh,tgv=1.);
  real HV=1.e20;
  Vh lb=Onb*g - (1.-Onb)*HV,ub=Onb*g + (1.-Onb)*HV;
  Vh u=0;
  IPOPT(A,dA,d2A,u[],lb=lb[],ub=ub[]);
  Th = adaptmesh(Th,u);
  plot(u,dim=3,fill=1,cmm="last");
```

```
  mesh3 Th3 = movemesh23(Th,transfo=[x,y,u(x,y)]);
  medit("final",Th3);
}
```

B.2 Problème des trois ellipses

Le script est scindé en deux fichiers `NSI-3e.edp` et `NSI-func.idp`. Le fichier avec l'extension `idp` est une sorte de fichier d'en-tête comportant des fonctions appelées dans le fichier `edp`. Il s'agit d'une fonction construisant le maillage et d'une méthode de résolution des équations de Navier-Stokes incompressibles par la méthode des caractéristiques.

```
/*********************************************
                  NSI-func.idp
*********************************************/

func int Showp(real[int] & p)
{
  for(int i=0;i<p.n;++i)
    cout << " " << p[i];
  return 0;
}
 func bb=[[-20,-15],[+20,15]];

 func real[int] QunPe(real aaa,real bbb,int n,real & l)
{
  real a = aaa ^ (-0.5);
  real b = bbb ^ (-0.5);
  real[int]  aa(n+1);
  aa=0.;
  real p= 2*pi/n;
  for(int i0 =0;i0< n;++i0)
  {
    int i1= i0+1;
    real t0=i0*p,t1=i1*p;
    real d0= a*(cos(t0)-cos(t1)), d1= b*(sin(t0)-sin(t1));
    real d= sqrt(d0*d0+d1*d1);
    aa[i1]= aa[i0]+d;
  }
  l= aa[n];
  aa *=  2*pi/l;
  return aa;
}

func real AbsCurve(real t,real [int] & a)
{
  int n = a.n-1,kk=0;
  real p= 2.*pi/n;
  int  i0=0,i1=n;
  if(t<a[i0]  t>a[i1]) return t;
  while ( i0+1 < i1)
```

```
40    {
41      int im = (i0+i1)/2;
42      if (t < a[im]) i1=im;
43      else i0=im;
44      assert(kk++<100);
45    }
46    real d = (a[i0]-t) / ( a[i0]-a[i1]);
47    return ((i0*(1-d) + i1*d))*p;
48  }
49
50
51  macro bee(i,eee,ddd,lab5)
52  border eee#i(tt = 2*pi, 0){
53    real t=AbsCurve(tt,aaa#i);
54    real  tx= (-l#i*sin(t))*re#i#x + (h#i*cos(t))*re#i#y;
55    real  ty=-(-l#i*sin(t))*re#i#y + (h#i*cos(t))*re#i#x ;
56    real tt=sqrt(square(tx)+square(ty));
57    tx/=tt;
58    ty/=tt;
59    x = (l#i*cos(t))*re#i#x + (h#i*sin(t))*re#i#y + x#i - ty*  ddd;
60    y = -(l#i*cos(t))*re#i#y + (h#i*sin(t))*re#i#x + y#i + tx* ddd;
61    label=lab5;
62  }
63  //EOM
64
65
66  macro mdist(i)
67  func real dist#i(real X,real Y) {
68    real xx=(X-x#i), yy= (Y-y#i);
69    real xxx= re#i#x*xx -re#i#y*yy;
70    real yyy= re#i#y*xx +re#i#x*yy;
71    /* xxx= l*cos(t)*l#i,
72       yyy= l*sin(t)*h#i,
73        in the ellipse axis repere */
74    real l=sqrt(square(xxx/l#i)+square(yyy/h#i));
75    xxx/=l; yyy/=l;
76    /*  xxx =  cos(t)*l#i  , ...*/
77    real xa =  (xxx)*re#i#x + (yyy)*re#i#y + x#i ;
78    real ya = -(xxx)*re#i#y + (yyy)*re#i#x + y#i ;
79    return sqrt(square(xa-X)+square(ya-Y));
80  } //EOM
81
82  func mesh MeshRef(real [int] param0,real[int]  pconst)
83  {
84    // ellipse properties
85    real l1 = pconst[0];   // length
86    real h1 = pconst[1];;    // height
87    real l2 = pconst[2];;
88    real h2 = pconst[3];;
```

```
real l3 = pconst[4];;
real h3 = pconst[5];;
int n = pconst[6];
int nb = pconst[7];
real [int] param=param0;
if(verbosity>2) cout << param << endl;
real x1= param[0];
real y1= param[1];
real a1= param[2];
real x2=0.;
real y2=0.;
real a2=0;
real x3= param[3];
real y3= param[4];
real a3= param[5];
real d2rad=pi/180.;
real re1x = cos(a1*d2rad);
real re1y = sin(a1*d2rad);
real re2x = cos(a2*d2rad);
real re2y = sin(a2*d2rad);
real re3x = cos(a3*d2rad);
real re3y = sin(a3*d2rad);
real vel = 1.0;
real le1,le2,le3;
real[int] aaa1=QunPe(l1,h1,100,le1);
real[int] aaa2=QunPe(l2,h2,100,le2);
real[int] aaa3=QunPe(l3,h3,100,le3);
real ddd=-h1/4;
real dddd=-h1/2;
bee(1,eee,ddd,5)
bee(2,eee,ddd,5)
bee(3,eee,ddd,5)
bee(1,eeee,dddd,5)
bee(2,eeee,dddd,5)
bee(3,eeee,dddd,5)
// three ellipses
border e1(tt = 2*pi, 0) {
      real t=AbsCurve(tt,aaa1);
      x = (l1*cos(t))*re1x + (h1*sin(t))*re1y + x1;
      y = -(l1*cos(t))*re1y + (h1*sin(t))*re1x + y1;
      label=3;
}
border e2(tt = 2*pi, 0) {
      real t=AbsCurve(tt,aaa2);
      x = (l2*cos(t))*re2x + (h2*sin(t))*re2y + x2;
      y = -(l2*cos(t))*re2y + (h2*sin(t))*re2x + y2;
      label=3;
}
border e3(tt = 2*pi, 0) {
```

```
138         real t=AbsCurve(tt,aaa3);
139         x = (l3*cos(t))*re3x + (h3*sin(t))*re3y + x3;
140         y = -(l3*cos(t))*re3y + (h3*sin(t))*re3x + y3;
141         label=3;
142   }
143   mesh The = buildmesh (e1(n) + e2(n*le2/le1) + e3(n*le3/le1)
144     +eee1(n) + eee2(n*le2/le1) + eee3(n*le3/le1)
145     +eeee1(-n) + eeee2(-n*le2/le1) + eeee3(-n*le3/le1));
146   return The;
147 }
148
149
150
151 func mesh ExternMesh(real [int] param,real[int]  pconst)
152 {
153   // ellipse properties
154   real l1 = pconst[0];   // length
155   real h1 = pconst[1];    // height
156   real l2 = pconst[2]
157   real h2 = pconst[3];
158   real l3 = pconst[4];
159   real h3 = pconst[5];
160   int n = pconst[6];
161   int nb = pconst[7];
162   // ----------------------
163   real s1 = 160;
164   real s2 = 320;
165   real X0= -80, Y0=-80;
166   real X1 = X0 + s2, Y1= Y0+ s1;
167   real x1= param[0];
168   real y1= param[1];
169   real a1= param[2];
170   real x2=0.;
171   real y2=0.;
172   real a2=0;
173   real x3= param[3];
174   real y3= param[4];
175   real a3= param[5];
176   real d2rad=pi/180.;
177   real re1x = cos(a1*d2rad);
178   real re1y = sin(a1*d2rad);
179   real re2x = cos(a2*d2rad);
180   real re2y = sin(a2*d2rad);
181   real re3x = cos(a3*d2rad);
182   real re3y = sin(a3*d2rad);
183   real vel = 1.0;
184   real le1,le2,le3;
185   real[int] aaa1=QunPe(l1,h1,100,le1);
186   real[int] aaa2=QunPe(l2,h2,100,le2);
```

```
187   real[int] aaa3=QunPe(l3,h3,100,le3);
188   real ddd=-h1/4;
189   real dddd=-h1/2;
190   bee(1,eee,ddd,5)
191   bee(2,eee,ddd,5)
192   bee(3,eee,ddd,5)
193   bee(1,eeee,dddd,5)
194   bee(2,eeee,dddd,5)
195   bee(3,eeee,dddd,5)
196   // boundary definitions
197   border G1(t=Y1,Y0){ x=X0; y=t; label=1; };
198   border G2(t=X0,X1){ x=t; y=Y0; label=1; };
199   border G3(t=Y0,Y1){ x=X1; y=t; label=2; };
200   border G4(t=X1,X0){ x=t; y=Y1; label=1; };
201   func bord = G1(nb) + G2(nb*s2/s1) + G3(nb) +
202               G4(nb*s2/s1) + eeee1(n) +
203               eeee2(n*le2/le1) + eeee3(n*le3/le1);
204   mesh tha=buildmesh(bord);
205   return tha;
206 }
207
208 func int NavierStokes
209 (real rey, mesh &The, mesh &thp, real[int] & ponthe,
210  real[int] & param, real[int] & param0,
211  real[int] & pconst, int iter)
212 {
213   int pplot =max(verbosity-1,0);
214   real d2rad=pi/180.;
215   real aaa=pconst[8]*d2rad;
216   //  infty velocity ...
217   real u1infty= cos(aaa);
218   real u2infty=sin(aaa);
219   real[int]  TF1 = [ param[0] , param[1],
220                      cos( (param[2]-param0[2] )* d2rad),
221                      sin( (param[2]-param0[2] )* d2rad) ];
222   real[int]  TF2 = [ param[3] , param[4],
223                      cos( (param[5]-param0[5] )* d2rad),
224                      sin( (param[5]-param0[5] )* d2rad) ];
225   real x1=  param0[0];
226   real y1 = param0[1];
227   real x2 = param0[3];
228   real y2 = param0[4];
229   if(verbosity>2)
230   {
231     cout << TF1 << endl;
232     cout << TF2 << endl;
233   }
234   real ll0=-0.5-pconst[2];
235   real ll1= 0.5+pconst[2];
```

```
236  func Fx =
237    x<ll0 ? ( TF1[0] + TF1[2]*(x-x1) + TF1[3]*(y-y1) )
238          : (x> ll1 ? ( TF2[0] + TF2[2]*(x-x2) + TF2[3]*(y-y2) ): x );
239  func Fy =
240    x<ll0 ? ( TF1[1] - TF1[3]*(x-x1) + TF1[2]*(y-y1) )
241          : (x> ll1 ? (  TF2[1] - TF2[3]*(x-x2) + TF2[2]*(y-y2) ) : y );
242  if(0){
243    mesh Th=The;
244    fespace Vh(Th,P1);
245    Vh u1=Fx-x, u2=Fy-y;
246    plot([u1,u2],wait=1);
247  }
248  mesh thf = movemesh(The,[Fx,Fy]);
249  thp=thf;
250  mesh tha = ExternMesh(param,pconst);
251  mesh th = tha+thf;
252  func bb=[[-20,-15],[+20,15]];
253  int[int] lredges=[3,5];
254  if(verbosity>3) plot(th,wait=1,bb=bb);
255  fespace Xh(th,P2);
256  fespace Mh(th,P1);
257  fespace Mhf(thf,P1);
258  fespace XXMh(th,[P2,P2,P1]);
259  XXMh [u1,u2,p];
260  XXMh [v1,v2,q];
261  macro div(u1,u2) (dx(u1)+dy(u2))//
262  macro grad(u) [dx(u),dy(u)]//
263  macro ugrad(u1,u2,v) (u1*dx(v)+u2*dy(v)) //
264  macro Ugrad(u1,u2,v1,v2) [ugrad(u1,u2,v1),ugrad(u1,u2,v2)]//
265  // solve the Stokes equation
266  solve Stokes ([u1,u2,p],[v1,v2,q],solver=UMFPACK) =
267    int2d(th)( ( dx(u1)*dx(v1) + dy(u1)*dy(v1)
268             +  dx(u2)*dx(v2) + dy(u2)*dy(v2) )
269             - p*q*(0.000001)
270             - p*div(v1,v2) - q*div(u1,u2)
271            )
272  + on(1,u1=u1infty,u2=u2infty)
273  + on(3,u1=0,u2=0);
274  if(verbosity>2)
275    plot(coef=0.05,cmm="Stokes equation",[u1,u2],fill=0,wait=0,bb=bb);
276  real re=min(rey,1.);
277  real nu=1./re;
278  XXMh [up1,up2,pp];
279  varf vDNS ([u1,u2,p],[v1,v2,q]) =
280    int2d(th)(
281      + nu * ( dx(u1)*dx(v1) + dy(u1)*dy(v1)
282      +  dx(u2)*dx(v2) + dy(u2)*dy(v2) )
283      - p*q*(0.000001)
284      - p*dx(v1)+ p*dy(v2)
```

```
285       - dx(u1)*q+ dy(u2)*q
286       + Ugrad(u1,u2,up1,up2)'*[v1,v2]
287       + Ugrad(up1,up2,u1,u2)'*[v1,v2]
288     )
289   + on(1,3,u1=0,u2=0);
290
291   varf vNS ([u1,u2,p],[v1,v2,q]) =
292     int2d(th)(
293       + nu * ( dx(up1)*dx(v1) + dy(up1)*dy(v1)
294       + dx(up2)*dx(v2) + dy(up2)*dy(v2) )
295       - pp*q*(0.000001)
296       - pp*dx(v1)+ pp*dy(v2)
297       - dx(up1)*q+ dy(up2)*q
298       + Ugrad(up1,up2,up1,up2)'*[v1,v2]//'
299     )
300   + on(1,3,u1=0,u2=0);
301   int i;
302   int ii = 0;
303   int err=0;
304   real wl2=0;
305   if(verbosity)
306     cout << "   nv " << th.nv << "  nt "
307          << th.nt << " ndof " << Xh.ndof << endl;
308   while(1)
309   {
310     ++ii;
311     re=min(re,rey);
312     real cpu1=clock();
313     // mesh adaptation
314     tha=adaptmesh(tha,[u1,u2],p,err=0.1,ratio=1.3,
315                   nbvx=100000,hmin=0.03,requirededges=lredges);
316     th = tha+ thf;
317     if(pplot) plot(th,bb=bb);
318     [u1,u2,p]=[u1,u2,p];
319     [up1,up2,pp]=[up1,up2,pp];
320     if(verbosity)
321       cout << "   nv " << th.nv << "  nt "
322            << th.nt << " ndof " << Xh.ndof << endl;
323     real[int] b(XXMh.ndof),w(XXMh.ndof);
324     // solve steady-state NS using Newton method
325     int kkkk=3;
326     for (i=0;i<=15;i++)
327     {
328           if  (i%kkkk==1)
329       {
330         kkkk*=2;
331         tha=adaptmesh(tha,[u1,u2],p,err=0.1,ratio=1.3,
332                       nbvx=100000,hmin=0.01,requirededges=lredges);
333         th = tha+ thf;
```

```
334         if(pplot>2) plot(th,bb=bb);
335         [u1,u2,p]=[u1,u2,p];
336         [up1,up2,pp]=[up1,up2,pp];
337         b.resize(XXMh.ndof);
338         w.resize(XXMh.ndof);
339           }
340       nu =1./re;
341       up1[]=u1[];
342       b = vNS(0,XXMh);
343       matrix Ans=vDNS(XXMh,XXMh); // build sparse matrix
344       set(Ans,solver=UMFPACK);
345       w = Ans^-1*b; // solve sparse matrix
346       u1[] -= w;
347       if(verbosity>1)
348         cout << " **iter = " << i << "  " << w.l2 << endl;
349       wl2=w.l2 ;
350       if(wl2<1e-8) break;
351       if(wl2>1e5)
352       {
353         cout << "WARNING: in NavierStokes wl2>1e5" << endl;
354         err=1; break;
355       }
356           plot(coef=0.02,
357                cmm="rey="+re+", iter = "+i+", w.l2 = "+w.l2+", th.nv = "+th.nv,
358                [u1,u2],p,fill=0,wait=0,nbiso=20,value=0,bb=bb);
359     }
360     if(err) break;
361     plot(cmm="rey="+re+" [u1,u2] et p  ",p,[u1,u2],fill=0,wait=1,
362          nbiso=20,value=0,ps="result-Re"+re+".eps",bb=bb);
363     cout << re << " " << rey << " NS " << i << " wl2= " << wl2
364          << "  "  << ii << ": " << clock() - cpu1 << " s" ;
365     Showp(param);
366     cout << endl;
367     if(re >= rey) break;
368     re*=2;
369   }
370   if(err==0)
371   {
372     int[int] fforder2=[1,1];
373     Mhf pf=p;
374     ponthe=pf[];
375   }
376   return err;
377 }
```

```
1 /*****************************************
2                  NSI-3e.edp
3 *****************************************/
4
5 load "CMA_ES_MPI"
```

```
6
7  include "NSI-func.idp"
8  real Reynolds=400.;
9  int www=0;
10 randinit(131);
11 verbosity = 0;
12 //   l1,h1, l2,h2, l3, h3, nb point ellp 1, nb point infini , angle
13 real[int]  pconst = [2,0.5,10,1,3.5,0.5,75,10,3] ;
14 real jold;
15
16 real[int] pmin = [ -20, -3., -10, 14.5, -3, 0];
17 real[int] pmax = [ -13. , 0  , 0  , 20  , 0   , 10];
18 real[int]  param0= pmin*0.5 + pmax*0.5;
19 real[int]  paramr = [-16,-1.5,-5,17,-1.5,5.];
20 real[int]  difff= paramr-param0;
21 real rhobb= difff.linfty;
22 cout << " rhobb = " << rhobb << endl;
23 mesh The = MeshRef(param0,pconst);
24 mesh Thr = The;
25 fespace Phe(The,P1);
26
27 func real[int] randparam()
28 {
29   real[int] res(6);
30   for(int i=0;i<6;++i)
31   {
32     real alpha = randreal1();
33     res[i] = pmin[i] + alpha*(pmax[i]-pmin[i]);
34   }
35   return res;
36 }
37 paramr = randparam();
38
39 Phe pr,p;
40 verbosity=0;
41 // build ref solution ...
42 plot(The,bb=bb,wait=0);
43
44
45 NavierStokes(Reynolds,The,Thr,pr[],paramr,param0, pconst,0);
46
47 plot(pr,dim=3,fill=1,value=1,wait=1,cmm="The reference pressure");
48 real  res =  int1d(The,3)( square(pr) );
49 cout << " pr.sum = " << pr[].sum << endl;
50
51 cout << " ||p||_L2^2  " << res << endl;
52
53 int iter =0;
54
```

```
55
56
57 func real J(real[int] & param)
58 {
59   int vv=verbosity;
60   mesh Thp;
61   real[int] pa=param;
62   for (int i=0;i<pa.n;i++)
63   {
64     pa[i]=min(pa[i],pmax[i]);
65     pa[i]=max(pa[i],pmin[i]);
66   }
67   real[int] dif=pa-param;
68   real ll= dif.l2^2 ;
69   NavierStokes(Reynolds,The,Thp,p[],pa,param0, pconst,iter++);
70   real  res =  int1d(The,3)( square(pr-p) );
71   cout << " J: " << iter ;
72   for(int i=0;i<pa.n;++i)
73     cout << " " << pa[i] ;
74   real[int] dd=param-paramr;
75   real distr=dd.linfty;
76   cout << " /  " <<     res  << " ******** + " << ll << " || pa-pr ||= "
77        << distr  << " " << p[].sum << " " << pr[].sum << endl;
78   verbosity=vv;
79   jold=res;
80   plot(pr,Thp,wait=www,dim=3,fill=1,cmm="diff  J " );
81   return res+ll*100;
82 }
83
84
85 ofstream jfile("J.dat");
86 real[int] pa=param0;
87
88 pa = randparam();
89
90 real mincost=cmaespMPI(J,pa,popsize=12);
91 cout  << "  at :" << pa << endl;
92 difff = pa -paramr;
93 cout << " diff " ;
94   for(int i=0;i<pa.n;++i)
95     cout << " " << difff[i] ;
96     cout << endl;
```

B.3 Simulation d'un condensat de Bose-Einstein

```
1 int wait=0;
2 real Rtf = 3.4;
3 real Rmax= 1.5 * Rtf;
```

```
real g = 1000;
real Omega = 0;
func Vtrap = (x*x+y*y)*0.5 +square(x*x+y*y)*0.25;
func r2 = x*x + y*y;
func Vtr  = r2*0.5 +square(r2)*0.25;
func dTr2 = 0.5 *(1. + r2);

real Omega0=1,Omega1=3, dOmega=0.1;
bool sameth0=0;//  in the loop of Omega using same inital mesh
int nbadap=2;// nb of adapt loop  0 => no adapt..
int nbopt=10, maxiter=10;
real tol=1e-8;
bool cmu = 1;
int M=200;//  starting number of point on outer circle..
func Pk = P1;//FE order selction
border C(t=0,2*pi){ x= Rmax*cos(t);y=Rmax*sin(t); label=1;}
mesh Th=buildmesh(C(M));

real mu = 50;

func rhoTF = max(0., (mu - Vtrap)/g );

func real MuUpdate()
{
  for( int iter =0; iter < 20 ; ++ iter)
  {
      real nu = int2d(Th)(rhoTF) -1.  ;
      real dmu =  int2d(Th) (real( (mu - Vtrap) >0) )/g ;
      cout << " iter " << iter << " err = " <<nu << " " << mu << endl;
      mu -= nu / dmu ;
      if( abs(nu ) < 1e-15) break;
  }
  assert( abs(int2d(Th)(rhoTF) -1.) < 1e-10);
  return mu;
}


fespace Vh(Th,Pk);
fespace Vh2(Th,[Pk,Pk]);
macro Grad(u) [dx(u),dy(u)] //

func real J(real[int] & u)
{
  Vh2 [ur,ui] ; ur[]=u;
  real E = int2d(Th) (
       ( Grad(ur)'*Grad(ur) + Grad(ui)'*Grad(ui) ) * 0.5
     + Vtrap* ( ur*ur + ui * ui)
     + g* square( ur*ur + ui * ui) * 0.5
     + Omega*(  y *( - dx(ur)*ui + dx(ui)*ur )
```

```
53                + x *(   dy(ur)*ui - dy(ui)*ur ) )
54                );
55    return E;
56 }
57
58 func real[int] dJ(real[int] & u)
59 {
60   Vh2 [ur,ui] ; ur[]=u;
61   varf vdj([aa,bb],[vr,vi]) =
62      int2d(Th) (
63        ( Grad(ur)'*Grad(vr) +  Grad(ui)'*Grad(vi) )
64      + Vtrap* ( ur*vr + ui * vi)*2.
65      + g*  ( ur*vr + ui*vi) *( ur*ur + ui*ui)
66      + Omega*(  y *( - dx(ur)*vi + dx(ui)*vr )
67                + x *(   dy(ur)*vi - dy(ui)*vr ) )
68      + Omega*(  y *( - dx(vr)*ui + dx(vi)*ur )
69                + x *(   dy(vr)*ui - dy(vi)*ur ) )
70                );
71      real [int] du= vdj(0,Vh2);
72    return du ;
73 }
74
75 func real[int] Constraint(real[int] & u)
76 {
77   Vh2 [ur,ui] ; ur[]=u;
78   real[int] c(1);
79   c[0] = int2d(Th) ( ur*ur + ui * ui)  ;
80   return c;
81 }
82 matrix MC;
83 func matrix dConstraint(real[int] & u)
84 {
85   Vh2 [ur,ui] ; ur[]=u;
86   varf vc([aa,bb],[vr,vi]) = int2d(Th) ( ( ur*vr + ui * vi) *2.) ;
87   real[int,int] MF(1,u.n);
88   MF(0,:) = vc(0,Vh2);
89   MC=MF;
90 }
91
92 matrix Hessian;
93
94 func matrix hJ(real[int] & u,real obj,real[int] & l)
95 {
96   Vh2 [ur,ui] ; ur[]=u;
97   varf vhj([wr,wi],[vr,vi]) =
98      int2d(Th) (
99       + (obj*Vtrap+ l[0]) * ( wr*vr + wi*vi)*2.
100      + obj*(
101        ( Grad(wr)'*Grad(vr) +  Grad(wi)'*Grad(vi) )
```

```
102      //    D of  :  g*  ( ur*vr + ui * vi) *( ur*ur + ui * ui)
103       + g*      ( wr*vr + wi*vi) * ( ur*ur + ui*ui)
104       + g*2.*   ( ur*vr + ui*vi) * ( wr*ur + wi*ui)
105
106       + Omega*(  y *( - dx(wr)*vi + dx(wi)*vr )
107                + x *(   dy(wr)*vi - dy(wi)*vr ) )
108       + Omega*(  y *( - dx(vr)*wi + dx(vi)*wr )
109                + x *(   dy(vr)*wi - dy(vi)*wr ) )
110              ));
111   Hessian = vhj(Vh2,Vh2);
112   return Hessian;
113 }
114
115 load "ff-Ipopt"
116
117 Vh<complex> u0 = sqrt( rhoTF);
118 Vh ur = real(u0);
119 plot(ur,wait=wait);
120 real[int] c=[1];
121 real cost=1000;
122 Vh2 [uur,uui] = [ real(u0), imag(u0)] ;
123 Vh2 [uur0,uui0] ;
124 real[int] lambda=[1];
125 real cpu,cpu0;
126 int omegaiter=0;
127
128 for( Omega= Omega0; Omega <= Omega1; Omega+= dOmega)
129 {
130   int adaptlevel=0;
131   cpu0 = clock();
132   int ok=0;
133   mesh Th0= Th;
134   for(int kk=1,nok=0;kk<= nbopt; ++kk)
135   {
136     if(nbadap ){
137      Th=adaptmesh(Th,[uur,uui],err=0.05,nbvx=100000,hmax=Rmax*2*pi/M);
138      adaptlevel++;
139      MuUpdate();
140     }
141
142     [uur,uui] = [uur,uui] ;
143
144     int[int] II(Vh2.ndof),JJ=0:Vh2.ndof-1; II=0;
145     ok=IPOPT(J,dJ,hJ,Constraint,dConstraint,uur[],
146              clb=c,cub=c,structjacc=[II,JJ],tol=tol,
147              maxiter=maxiter*(1+adaptlevel),objvalue=cost,
148              lm=lambda,warmstart=(kk>1));
149     cout << kk << " IPOPT = " << ok << " J = " << cost << " Th.nv =" << Th.nv;
150     cout << " Omega =" << Omega << " ok =" << ok <<  endl;
```

```
151     nok += (ok==0);
152     if( ok==0)
153     {
154      Vh uu= sqrt(square(uur)+square(uui));
155      plot(uu,cmm=" Omega ="+Omega+" kk "+kk+" J ="+cost+"  ok ="+ok ,dim=3);
156     }
157     if(nbadap && (nbadap==0)  nok >= nbadap) break;
158   }
159   cpu=clock();
160   cout << "************************************************" << endl;
161   cout << " ---- IPOPT = " << ok << " J = " << cost << " Th.nv =" << Th.nv;
162   cout << " Omega =" << Omega << " cpu ="  << cpu-cpu0 << " s" <<  endl;
163
164   Vh uu= sqrt(square(uur)+square(uui));
165   Vh<complex> cuu = uur + uui*(1i);
166   Vh phase = arg(cuu);
167   Vh2 [vx,vy] = [uur*dx(uui)-uui*dx(uur),uur*dy(uui)-uui*dy(uur)];
168   plot(uu,cmm="Density - Omega =" + Omega+  " J =" + cost + "  ok =" + ok ,dim=3);
169   plot(phase,cmm="Phase - Omega =" + Omega+  " J =" + cost + "  ok =" + ok );
170   plot([vx,vy],cmm="Probability stream - Omega =" + Omega+  " J =" + cost + "
ok =" + ok);
171   if( sameth0 )
172   {
173     Th=Th0;
174     [uur,uui] = [uur,uui] ;
175   }
176   omegaiter++;
177 }
```

B.4 Problème de contact

Le code présenté ci-dessous résout numériquement le problème suivant, dérivé du problème de contact avec conditions aux bords de Signorini analysé dans [7] : étant donnés deux domaines Ω^s et Ω^i de $\mathbb{R}^2$, tangents dont la frontière commune est notée $\Gamma_C = \overline{\Omega^s} \cap \overline{\Omega^i}$. On note également $\Gamma = \partial\left(\Omega^s \cup \Omega^i\right)$ et $\Gamma_n^\alpha = \Gamma \cap \overline{\Omega^\alpha}$ (pour $\alpha = s$ ou i).
Trouver deux fonctions $p^s \in H^1(\Omega^s)$ et $p^i \in H^1(\Omega^i)$ telles que :

$$(p^s, p^i) = \underset{(u^s,u^i)\in K}{\operatorname{argmin}} \sum_{\alpha\in\{s,i\}} \frac{1}{2}\int_{\Omega^\alpha} |\nabla u^\alpha|^2 - \int_{\Omega^\alpha} f^\alpha u^\alpha - \int_{\Gamma_n^\alpha} g_n^\alpha u^\alpha$$

où :

$$K = \left\{ (u^s, u^i) \in H^1(\Omega^s) \times H^1(\Omega^i) \;\middle|\; \begin{array}{c} u^\alpha - g_d^\alpha \in H_0^1(\Omega^\alpha)\ ,\ \alpha = s,\ i \\ \text{et} \\ (u^s - u^i)|_{\Gamma_C} \geq 0 \end{array} \right\}$$

Les fonctions g_d^α, f^α et g_n^α sont des données du problème. Dans le code, elles sont choisies de sorte à ce qu'une solution analytique soit connue, et de manière à pouvoir calculer les erreurs *a posteriori* et pouvoir ainsi observer les courbes d'évolution de cette erreur en fonction de la finesse de la triangulation dans diverses situations (maillage conforme, non conforme, second membre polynomial, ou non polynomial, *etc...*). On utilise IPOPT pour la résolution de ce problème d'optimisation quadratique avec contraintes d'inégalité linéaires. La méthode s'est montrée très efficace dans les situations abordées.

```
load "ff-Ipopt"
real Xm=-1,XM=1;
real Y0=0,YM=1,Ym=-1;
int[int] lu=[2,3,1,1], lb=[1,3,2,1],lm=[2,1,4,1];

for(int nnn=20;nnn<=21;nnn+=2)
{
int nn=nnn,nu=nnn,nb=nnn;
int nc=1;//0:maillage conforme - 1:non conforme
mesh Thu=square(nn,nu,label=lu,[Xm + x*(XM-Xm), Y0 + y*(YM-Y0)]);
mesh Thb=square(nn-nc,nb,label=lb,[Xm + x*(XM-Xm), Ym + y*(Y0-Ym)]);
mesh Thm=square(nn,1,label=lm,[Xm + x*(XM-Xm), Y0 + y*(0.1-Y0)]);
fespace Vu(Thu,P2), Vb(Thb,P2), Vm(Thm,P2);

macro xplus(x)  ((x)<0.? 0 : x)//
macro xmoins(x) ((x)>0.? 0 : x)//
macro Grad(u) [dx(u),dy(u)]//EOM

ofstream data("data_nc.txt",append);

real cc=1.;
real p2=pi*0.5,p2s=square(pi*0.5);
func fa = 0;
func ga = (y-1)*y +x*(1-x);
func gax = 1-2*x ;
func gay = 2*y-1 ;
func gu  =
  ga  + (x>0 ? x*sin(p2*x) : -sin(pi*y)/2*x*sin(p2*x)  )  ;
func gux =  gax
  + (x>0 ?  (sin(p2*x)+ p2*x*cos(p2*x) )
         :  -sin(pi*y)/2 * (sin(p2*x)+ p2*x*cos(p2*x) ));
func guy =  gay + (x>0 ?  0 :   -pi*cos(pi*y)/2*x*sin(p2*x))  ;

func gb  =
  ga  +  (x>0 ? -x*sin(p2*x) : -sin(pi*y)/2*x*sin(p2*x)  )  ;
func gbx =  gax
  + (x>0 ? - (sin(p2*x)+ p2*x*cos(p2*x) )
         :  -sin(pi*y)/2 * (sin(p2*x)+ p2*x*cos(p2*x) ));
func gby =  guy ;
func g= (y <0)?gb : gu;

func fu  =  fa
  + (x>0 ?   -2*p2*cos(p2*x) + x*p2s*sin(p2*x)
         :  -sin(pi*y)/2 * (-2*p2*cos(p2*x)+x*p2s*sin(p2*x))
            -square(pi)*sin(pi*y)/2*x*sin(p2*x)) ;
func fb  =  fa
  + (x>0 ?   2*p2*cos(p2*x) - x*p2s*sin(p2*x)
         :  -sin(pi*y)/2 * (-2*p2*cos(p2*x)+x*p2s*sin(p2*x))
            -square(pi)*sin(pi*y)/2*x*sin(p2*x)) ;
```

```
50 func f=0;
51
52 int iter=0,giter=0,bcgiter=0;
53
54
55 varf vu(u,v) =
56     int2d(Thu)(0.5*dx(u)*dx(v) + 0.5*dy(u)*dy(v))
57   + int1d(Thu,2)(gay*v) - int2d(Thu)( fu*v);
58 varf vb(u,v) =
59     int2d(Thb)( 0.5*dx(u)*dx(v) + 0.5*dy(u)*dy(v))
60   - int1d(Thb,2)(gay*v) - int2d(Thb)( fb*v) ;
61 varf dvu(u,v) = int2d(Thu)( dx(u)*dx(v) + dy(u)*dy(v));
62 varf dvb(u,v) = int2d(Thb)( dx(u)*dx(v) + dy(u)*dy(v));
63
64
65 Vu Uu,V;
66 Vb Ub,v;
67 solve Poissu(Uu,V) =
68     int2d(Thu)( Grad(Uu)'*Grad(V) )
69   - int2d(Thu)( fu*V) +int1d(Thu,2)(guy*V) +on(1,3,Uu=gu);
70 solve Poissb(Ub,v) =
71     int2d(Thb)( Grad(Ub)'*Grad(v) )
72   - int2d(Thb)( fb*v) -int1d(Thb,2)(gby*v)+on(1,3,Ub=gb);
73 plot(Uu,Ub,wait=1,fill=1,value=1,dim=3,cmm="separate lineare problems");
74
75
76 matrix Au = vu(Vu,Vu,solver=CG);
77 matrix Ab = vb(Vb,Vb,solver=CG);
78 matrix dAu = dvu(Vu,Vu,solver=CG);
79 matrix dAb = dvb(Vb,Vb,solver=CG);
80 real[int] bu = vu(0,Vu), bb = vb(0,Vb);
81
82 varf bord(u,v) = on(1,3,u=1);
83 Vu Bordu,Inu,BCu;
84 Bordu[] = bord(0,Vu,tgv=1.);
85 Inu[] = Bordu[] ? 0.:1.;
86 BCu = Bordu*gu;
87 Vb Bordb,Inb,BCb;
88 Bordb[] = bord(0,Vb,tgv=1.);
89 Inb[] = Bordb[] ? 0.:1.;
90 BCb = Bordb*gb;
91 int[int] IndIu(Inu[].sum),IndIb(Inb[].sum);
92 {
93   int k=0;
94   for(int i=0;i<Vu.ndof;++i) if(Inu[][i]==1.) {IndIu[k]=i; ++k;}
95   k=0;
96   for(int i=0;i<Vb.ndof;++i) if(Inb[][i]==1.) {IndIb[k]=i; ++k;}
97 }
98
```

```
matrix AAu = dAu(IndIu,IndIu);
matrix AAb = dAb(IndIb,IndIb);
matrix H = [ [AAu, 0 ] , [0 , AAb] ];

func int SPPToFEF(real[int] &fef1,real[int] &fef2,real[int] &ssp)
{
  int k=0;
  for(int i=0;i<Vu.ndof;++i)
  {
    if(Bordu[][i]==1.) fef1[i] = BCu[][i];
    else {fef1[i] = ssp[k]; ++k;}
  }
  for(int i=0;i<Vb.ndof;++i)
  {
    if(Bordb[][i]==1.) fef2[i] = BCb[][i];
    else {fef2[i] = ssp[k]; ++k;}
  }
  return 1;
}
func int FEFToSPP(real[int] &fef1,real[int] &fef2,real[int] &ssp)
{
  for(int k=0;k<Inu[].sum;++k) ssp[k] = fef1[IndIu[k]];
  for(int k=0;k<Inb[].sum;++k) ssp[k+Inu[].sum] = fef2[IndIb[k]];
  return 1;
}

{ //Exact solution
   Vu u1=gu;
   Vb u2=gb;
   real n2 = int2d(Thu)(square(u1-Uu)) + int2d(Thb)(square(u2-Ub));
   real ng = int2d(Thu)(square(dx(u1)-dx(Uu))+square(dy(u1)-dy(Uu))) +
             int2d(Thb)(square(dx(u2)-dx(Ub))+square(dy(u2)-dy(Ub)));
   real[int] d1 = u1[] - Uu[], d2 = u2[] - Ub[];
   d1=abs(d1);
   d2=abs(d2);
   real ninf = max(d1.max,d2.max);
   real[int] tab(u1.n+u2.n);
   cout << "L2="<< sqrt(n2) << " H=" << sqrt(ng) << " infty=" << ninf << endl;
   FEFToSPP(u1[],u2[],tab);
   plot(u1,u2,cmm="exact - L2="+n2+" H="+ng+"   infty="+ninf,
        wait=1,dim=3,fill=1,value=1);
}


func real J(real[int] &X)
{
  Vu U;
  Vb B;
  SPPToFEF(U[],B[],X);
```

```
148   real[int] AuU = Au*U[], AbB = Ab*B[];
149   AuU += bu;
150   AbB += bb;
151   real res = (U[] '* AuU) + (B[] '* AbB);
152   ++iter;
153   real nrm2 = int2d(Thu)(square(U-gu)) + int2d(Thb)(square(B-gb));
154   plot(U,B,dim=3,cmm="J = "+res+" / " + "||U-Ue||="+sqrt(nrm2)
155        +" / "+iter+" J evaluations"+" / " + giter
156        + " grad evaluations",fill=1,nbiso=20,value=1);
157   {
158     Vu u1=gu;
159     Vb u2=gb;
160     real n2 = int2d(Thu)(square(gu-U)) + int2d(Thb)(square(gb-B));
161     real ng = int2d(Thu)(square(gux-dx(U))+square(guy-dy(U))) +
162               int2d(Thb)(square(gbx-dx(B))+square(gby-dy(B)));
163     real[int] d1 = u1[] - U[], d2 = u2[] - B[];
164     d1=abs(d1);
165     d2=abs(d2);
166     real ninf = max(d1.max,d2.max);
167     cout << "Iter:" << iter << " - l2=" << sqrt(n2) << " - l2grad="
168          << sqrt(ng) << " - infty=" << ninf << endl;
169   }
170   return res;
171 }
172
173
174
175 func real[int] dJ(real[int] &X)
176 {
177   Vu U;
178   Vb B;
179   SPPToFEF(U[],B[],X);
180   real[int] Gu = dAu * U[];
181   real[int] Gb = dAb * B[];
182   Gu += bu;
183   Gb += bb;
184   ++giter;
185   real[int] grad(X.n);
186   FEFToSPP(Gu,Gb,grad);
187 }
188
189 func matrix HJ(real[int] &X) {return H;}
190 matrix Bu = interpolate(Vm,Vu,inside=0);
191 matrix Bb = interpolate(Vm,Vb,inside=0);
192 matrix iBu = interpolate(Vu,Vm,inside=0);
193 matrix iBb = interpolate(Vb,Vm,inside=1);
194 //matrix BBu,BBb;
195 int Nc=0;
196 int[int] I=0:Vm.ndof-1;
```

```
varf vm2(u,v) = on(2,u=1) + on(1,3,u=0);
real[int]  bc2= vm2(0,Vm,tgv=1);
Vm phic=1, phicc=0;// zone de contact .
int[int] ind(bc2.sum);
{
  int k=0;
  for(int i=0;i<bc2.n;++i)  if(bc2[i]) {ind[k]=i; ++k;}
}
matrix C;
{
  matrix BBu = Bu(ind,IndIu);
  BBu = - BBu;
  matrix BBb = Bb(ind,IndIb);
  C = [ [BBu , BBb] ];
}

func real[int] Jumps(real[int] &X)
{
  real[int] jumps = C*X;
  return jumps;
}

func matrix dJumps(real[int] &X) {return C;}

Vu su=gu;
Vb sb=gb;
real[int] start(Inu[].sum+Inb[].sum);
real[int] mini(start.n),maxi(start.n);
mini=-100;
maxi=100;
Uu=-1.;
Ub=1.;
FEFToSPP(Uu[],Ub[],start);
plot(su,sb,wait=1,cmm="exact solution");

real[int] Clb(C.n),Cub(C.n);
Clb=-1.e19;
Cub=0.;

//Resolution du probleme :
IPOPT(J,dJ,HJ,Jumps,dJumps,start,clb=Clb,cub=Cub);

func real Jff(real[int] &X)
{
  Vu U;
  Vb B;
  SPPToFEF(U[],B[],X);
  real[int] AuU = Au*U[], AbB = Ab*B[];
  AuU += bu;
```

```
246   AbB += bb;
247   real res = (U[] '* AuU) + (B[] '* AbB);
248   ++iter;
249   real nrm2 = int2d(Thu)(square(U-gu)) + int2d(Thb)(square(B-gb));
250   plot(U,B,dim=3,cmm="J = "+res+" / " + "||U-Ue||="+sqrt(nrm2)
251        +" / "+iter+" J evaluations"+" / " + giter
252        +" grad evaluations",wait=0,fill=1,nbiso=20,value=1);
253   {
254     Vu u1=gu;
255     Vb u2=gb;
256     real n2 = int2d(Thu)(square(gu-U)) + int2d(Thb)(square(gb-B));
257     real ng = int2d(Thu)(square(gux-dx(U))+square(guy-dy(U))) +
258               int2d(Thb)(square(gbx-dx(B))+square(gby-dy(B)));
259     real[int] d1 = u1[] - U[], d2 = u2[] - B[];
260     d1=abs(d1);
261     d2=abs(d2);
262     real ninf = max(d1.max,d2.max);
263     data << nn << " " << 1./nn << " " << iter << " " << sqrt(n2)
264          << " " << sqrt(ng) << " " << ninf << endl;
265     cout << nn << " " << 1./nn << " " << iter << " " << sqrt(n2)
266          << " " << sqrt(ng) << " " << ninf << endl;
267   }
268   return res;
269 }
270
271 cout << "Apres IPOPT : " << Jff(start) << endl;
272 }
```

Annexe C

Modèles de classes pour la différentiation automatique

Nous proposons ici quelques éléments de programmation d'un modèle de classe pour la différentiation automatique en mode direct, paramétré par le type des données utilisées pour représenter la valeur et les composantes du gradient. Il ne s'agit pas ici d'aborder les questions de performance mais seulement de montrer comment la programmation générique permet de définir une vaste palette d'outils avec un volume de code minime.

C.1 Pour le mode direct

C.1.1 La classe `ddouble`

```
template<typename T> class ddouble {
  public:
    typedef T value_type;
    static const int N = MaxNumberOfDerivatives;
    static int nd; //nombre de dérivées courant
  private:
    T x;
    T dx[N];
  public:
    //Declaration et initialisation d'un ddouble par une constante
    //les valeurs par defaut permettent de definir un constructeur par
        defaut
    ddouble(const T &c=0,bool init_dx=true) : x(c)
    {if(init_dx) for(int i=0;i<nd;++i) dx[i] = T(0);}
    //Pour initialiser par un ddouble avec type sous jacent different
        si
    //la conversion de celui-ci vers T est possible
    template<class K> explicit ddouble(const ddouble<K> &X) : x(X.val()
        )
    {
      static_assert(is_convertible<K,T>::value);
      for(int i=0;i<nd;++i) dx[i] = X.d(i);
    }

    //const accesseurs :
    const T& val() const {return x;}
    const T& d(int i) const {return dx[i];}
```

```
25      //non const accesseurs :
26      T& val() {return x;}
27      T& d(int i) {return dx[i];}
28
29      //affectation d'une valeur constante :
30      ddouble& operator=(const T &c)
31      {
32        x=c;
33        for(int i=0;i<nd;++i) dx[i] = T(0);
34        return *this;
35      }
36      //affectation d'un ddouble si les types sont compatibles
37      template<typename K> ddouble& operator=(const ddouble<K> &X)
38      {
39        static_assert(is_convertible<K,T>::value);
40        x = X.val();
41        for(int i=0;i<nd;++i) dx[i] = X.d(i);
42      }
43
44      //operateurs arithmetiques d'affectation
45      //(f+c)' = f' si c constante
46      ddouble& operator+=(const T &c) {x += c; return *this;}
47      template<typename K> ddouble& operator+=(const ddouble<K> &X)
48      {
49        static_assert(is_convertible<K,T>::value);
50        for(int i=0;i<nd;++i) dx[i] += X.d(i);//(f+g)' = f' + g'
51        x += X.val();
52        return *this;
53      }
54      ddouble& operator*=(double c)
55      {
56        for(int i=0;i<nd;++i) dx[i] *= c;//(cf)'=cf' si c constante
57        x *= c;
58        return *this;
59      }
60      template<typename K> ddouble& operator*=(const ddouble<K> &X)
61      {
62        static_assert(is_convertible<K,T>::value);
63        for(int i=0;i<nd;++i) dx[i] = dx*X.val() + x*X.d(i);
64        x *= X.val();
65        return *this;
66      }
67      //declarations et definitions semblables pour les operateurs
            membres -=
68      //et /=
69      ...
70      //La fonction SetDiff
71      ddouble& SetDiff(int i)
72      {
73        for(int j=0;j<nd;++j) dx[j] = (i==j);
74        return *this;
75      }
76  };
```

C.1.2 Surcharge des opérateurs et des fonctions pour `ddouble`

Paramétrons dès maintenant la classe `common_type` comme en 4.2.2 pour la rendre compatible avec notre modèle de classes :

```
77 namespace std
78 {
79   template<typename T,typename K> class common_type<ddouble<T>,K>
80   {typedef typename ddouble<common_type<T,K>::type> type;};
81   template<typename T,typename K> class common_type<K,ddouble<T>>
82   {typedef typename ddouble<common_type<K,T>::type> type;};
83   template<typename T1,typename T2>
84   class common_type<ddouble<T1>,ddouble<T2>>
85   {typedef typename ddouble<common_type<T1,T2>::type> type;};
86 }
```

Exhibons ensuite quelques surcharges d'opérateurs, sans toutefois, ici encore, les faire toutes apparaître :

```
87 //Les surcharges de l'addition
88 template<class L,class R> typename common_type<ddouble<L>,R>::type
89 operator+(const ddouble<L> &l,const R &r)
90 {
91   typename common_type<ddouble<L>,R>::type res(l);
92   return (res += r);
93 }
94 template<class L,class R> typename common_type<L,ddouble<R>>::type
95 operator+(const L &l,const ddouble<R> &r)
96 {
97   typename common_type<ddouble<L>,R>::type res(r);
98   return (res += l);
99 }
100 template<class L,class R> ddouble<typename common_type<L,R>::type>
101 operator+(const ddouble<L> &l,const ddouble<L> &l)
102 {
103   typedef ddouble<typename common_type<L,R>::type> return_type;
104   return_type res(l.val()+r.val(),false);
105   for(int i=0;i<return_type::nd;++i) res.d(i) = l.d(i)+r.d(i);
106   return res;
107 }
108
109 //les surcharges de la multiplication
110 template<class L,class R> typename common_type<ddouble<L>,R>::type
111 operator*(const ddouble<L> &l,const R &r)
112 {
113   typedef typename common_type<ddouble<L>,R>::type return_type;
114   return_type res(l.val()*r,false);
115   for(int i=0;i<return_type::nd;++i) res.d(i) = l.d(i)*r;
116   return res;
117 }
118 template<class L,class R> typename common_type<L,ddouble<R>>::type
119 operator*(const L &l,const ddouble<R> &r)
120 {
121   typedef typename common_type<ddouble<L>,R>::type return_type;
```

```
122   return_type res(l*r.val(),false);
123   for(int i=0;i<return_type::nd;++i) res.d(i) = l*r.d(i);
124   return res;
125 }
126 template<class L,class R> ddouble<typename common_type<L,R>::type>
127 operator*(const ddouble<L> &l,const ddouble<L> &l)
128 {
129   typedef ddouble<typename common_type<L,R>::type> return_type;
130   return_type res(l.val()*r.val(),false);
131   for(int i=0;i<return_type::nd;++i) res.d(i) = l.d(i)*r.val()+l.val()*
          r.d(i);
132   return res;
133 }
134
135 //etc...
136 ...
```

Le code à écrire pour la surcharge des fonctions ne diffère que très peu de 4.2.2 :

```
137 //Surcharge de la fonction racine carree
138 template<class T> ddouble<T> sqrt(const ddouble<T> &X)
139 {
140   const T sqrtx =
          sqrt(X.val());//pour \'eviter de le calculer N+1 fois
141   ddouble<T> Y(sqrtx,false);//inutile d'initialiser dx
142   for(int i=0;i<ddouble<T>::nd;++i) Y.d(i) = X.d(i)/(2.*sqrtx);
143   return Y;
144 }
145 //Surcharge de la fonction cosinus
146 template<class T> ddouble<T> cos(const ddouble<T> &X)
147 {
148   ddouble Y( cos(X.val()),false);
149   const T msinx = -sin(X.val());
150   for(int i=0;i<ddouble<T>::nd;++i) Y.d(i) = X.d(i)*msinx;
151   return Y;
152 }
153 //Declarations et definitions semblables pour toute autre fonction
      mathematique de cmath
154 ...
```

Donnons enfin un exemple d'opérateur logique, dont chacun demandera trois surcharges pour permettre la comparaison entre `ddouble` de types internes non uniformes :

```
155 //exemple avec <
156 template<class L,class R>
157 inline bool operator<(const ddouble<L> &l,const R &r)
158 {return l.val() < r;}
159 template<class L,class R>
160 inline bool operator<(const L &l,const ddouble<R> &r)
161 {return l < r.val();}
162 template<class L,class R>
163 inline bool operator<(const ddouble<L> &l,const ddouble<R> &r)
```

```
164 {return l.val() < r.val();}
165 ...
```

C.2 Listing des classes utilisées dans FreeFem++

C.2.1 Modèle de classe du prototype basé sur FreeFem++-3.3

```
1  #ifndef _STATIC_FAD_
2  #define _STATIC_FAD_
3
4  #include <cassert>
5  #include <iostream>
6  #include <fstream>
7  #include <complex>
8  #include <typeinfo>
9  #include "config-wrapper.h"
10
11 #define NDIFF 50
12
13
14 template<class T,int ND> class FAD;
15
16
17 template<class T> struct NumericType {typedef T value_type;};
18 template<class T> struct NumericType< std::complex<T> >
19   {typedef T value_type;};
20
21 ...
22
23 template<class T,int ND> class FAD
24 {
25   public:
26     static const int nd=ND;
27     struct WithoutConstructor{T x,dx[ND];};
28     typedef T type;
29     typedef typename NumericType<T>::value_type value_type;
30
31   private:
32     T x,dx[ND];
33   public:
34     FAD(); //constructeur par défaut
35     FAD(const T&,bool init=true);
36     FAD(const T&,const T*);
37     FAD(const FAD&);//copy constructor
38     FAD(const WithoutConstructor&);
39
40     ~FAD() {}
41
42     T& operator[](int i) {return i==0 ? x : dx[i-1];}
43     const T& operator[](int i) const {return i==0 ? x : dx[i-1];}
44     T& val() {return x;}
45     const T& val() const {return x;}
46     T& d(int i=0) {return dx[i];}
```

```
47      const T & d(int i=0) const {return dx[i];}
48      static long size() {return ND;}
49      T* grad(){return dx;}
50      const T* grad()const{return dx;}
51
52      void SetDiff(int ith)
53      {
54        for(int i=0;i<ND;i++) dx[i]=T(0);
55        dx[ith]=T(1);
56      }
57      FAD& operator=(const WithoutConstructor&);
58      FAD& operator=(const FAD &);
59      FAD& operator+=(const FAD &);
60      FAD& operator-=(const FAD &);
61      FAD& operator*=(const FAD &);
62      FAD& operator/=(const FAD &);
63      FAD& operator=(const T &);
64      FAD& operator+=(const T &);
65      FAD& operator-=(const T &);
66      FAD& operator*=(const T &);
67      FAD& operator/=(const T &);
68      bool operator!()const{return !x;}
69
70      operator WithoutConstructor() const;
71      template<typename X> operator X() const {return static_cast<X>(x);}
72      template<typename X> operator FAD<X,ND>() const;
73      const FAD<value_type,ND>& real() const {return *this;}
74      FAD<value_type,ND> imag() const {return FAD<value_type,ND>(0);}
75      FAD<value_type,ND> norm() const {return x>0 ? *this : -*this;}
76      const FAD<value_type,ND>& conj() const {return *this;}
77  };
78
79  template<class T,int ND> // Specialisation pour les complexes
80  class FAD< std::complex<T>,ND>
81  {
82    public:
83      static const int nd = ND;
84      typedef  std::complex<T> type;
85      typedef  std::complex<T> C;
86      typedef T value_type;
87    private:
88      C x,dx[ND];
89    public:
90      FAD();
91      FAD(const T&,bool init=true);
92      FAD(const C&,bool init=true);
93      FAD(const C&,const C*);
94      FAD(const FAD<T,ND>&);
95      FAD(const FAD<T,ND>&,const FAD<T,ND>&);
96      FAD(const T&,const T*,const T&,const T*);
97      FAD(const FAD<C,ND>&);
98      ~FAD() {}
99
100
101     C& operator[](int i) {return i==0 ? x : dx[i-1];}
102     const C& operator[](int i) const {return i==0 ? x : dx[i-1];}
```

```
103     C& val() {return x;}
104     const C& val() const {return x;}
105     C& d(int i=0) {return dx[i];}
106     const C& d(int i=0) const {return dx[i];}
107     const long size()const{return ND;}
108     C* grad(){return dx;}
109     const C* grad()const{return dx;}
110
111     void SetDiff(int ith,bool im=0)
112     {
113       for(int i=0;i<ND;i++) dx[i]=C(0);
114       dx[ith]= im ? C(0.,1.) : C(1.);
115     }
116
117     FAD<C,ND>& operator=(const FAD<C,ND> &);
118     FAD<C,ND>& operator+=(const FAD<C,ND> &);
119     FAD<C,ND>& operator-=(const FAD<C,ND> &);
120     FAD<C,ND>& operator*=(const FAD<C,ND> &);
121     FAD<C,ND>& operator/=(const FAD<C,ND> &);
122     FAD<C,ND>& operator=(const FAD<T,ND> &);
123     FAD<C,ND>& operator+=(const FAD<T,ND> &);
124     FAD<C,ND>& operator-=(const FAD<T,ND> &);
125     FAD<C,ND>& operator*=(const FAD<T,ND> &);
126     FAD<C,ND>& operator/=(const FAD<T,ND> &);
127     FAD<C,ND>& operator=(const C &);
128     FAD<C,ND>& operator+=(const C &);
129     FAD<C,ND>& operator-=(const C &);
130     FAD<C,ND>& operator*=(const C &);
131     FAD<C,ND>& operator/=(const C &);
132     FAD<C,ND>& operator=(const T &);
133     FAD<C,ND>& operator+=(const T &);
134     FAD<C,ND>& operator-=(const T &);
135     FAD<C,ND>& operator*=(const T &);
136     FAD<C,ND>& operator/=(const T &);
137     bool operator!()const{return !x;}
138
139     template<typename X> operator FAD<X,ND>() const;
140
141     FAD<value_type,ND> real() const
142     {
143       FAD<value_type,ND> r( std::real(x),false);
144       for(int i=0;i<ND;i++) r.d(i) = std::real(dx[i]);
145       return r;
146     }
147     FAD<value_type,ND> imag() const
148     {
149       FAD<value_type,ND> r( std:: imag(x),false);
150       for(int i=0;i<ND;i++) r.d(i) = std::imag(dx[i]);
151       return r;
152     }
153     FAD<value_type,ND> norm() const;
154     FAD<C,ND> conj() const
155     {
156       FAD<C,ND> r( std::conj(x),false);
157       for(int i=0;i<ND;i++) r.d(i) = std::conj(dx[i]);
158       return r;
```

```
159     }
160 };
161
162 ////////////////////////////////////////////////////////
163 // Definition des membres de FAD<T,ND> - T non complexe //
164 ////////////////////////////////////////////////////////
165
166 template<class T,int ND> FAD<T,ND>::FAD() : x(0) , dx()
167   {for(int i=0;i<ND;i++) dx[i] = T(0);}
168
169 template<class T,int ND>
170 FAD<T,ND>::FAD(const T &y,bool init) : x(y),dx()
171   {if(init) for(int i=0;i<ND;i++) dx[i] = T(0);}
172
173 template<class T,int ND>
174 FAD<T,ND>::FAD(const T &y,const T *dy) : x(y),dx()
175 {
176   if(dy) for(int i=0;i<ND;i++) dx[i] = dy[i];
177   else for(int i=0;i<ND;i++) dx[i] = T(0);
178 }
179
180 template<class T,int ND>
181 FAD<T,ND>::FAD(const FAD<T,ND> &X) : x(X.x),dx()
182   {for(int i=0;i<ND;i++) dx[i] = X.dx[i];}
183
184 template<class T,int ND>
185 FAD<T,ND>::FAD(const WithoutConstructor &X) : x(X.x),dx()
186   {for(int i=0;i<ND;i++) dx[i]=X.dx[i];}
187
188
189
190 template<class T,int ND>
191 FAD<T,ND>& FAD<T,ND>::operator=(const WithoutConstructor &X)
192 {
193   for(int i=0;i<ND;i++) dx[i] = X.dx[i];
194   x = X.x;
195   return *this;
196 }
197
198 template<class T,int ND>
199 FAD<T,ND>& FAD<T,ND>::operator=(const FAD<T,ND> &X)
200 {
201   x = X.x;
202   for(int i=0;i<ND;i++) dx[i] = X.dx[i];
203   return *this;
204 }
205
206 template<class T,int ND>
207 FAD<T,ND>& FAD<T,ND>::operator=(const T &X)
208 {
209   x = X;
210   for(int i=0;i<ND;i++) dx[i]=T(0);
211   return *this;
212 }
213
214 template<class T,int ND>
```

```
215 FAD<T,ND>& FAD<T,ND>::operator+=(const FAD<T,ND> &X)
216 {
217   for(int i=0;i<ND;i++) dx[i] += X.dx[i];
218   x += X.x;
219   return *this;
220 }
221 template<class T,int ND>
222 FAD<T,ND>& FAD<T,ND>::operator+=(const T &X) {x += X; return *this;}
223
224 ... //operator-,*,/ - ommited
225
226 template<class T,int ND> FAD<T,ND>::operator WithoutConstructor()const
227 {
228   WithoutConstructor res;
229   res.x = x;
230   for(int i=0;i<ND;i++) res.dx[i] = dx[i];
231   return res;
232 }
233
234 ////////////////////////////////////////////////
235 // Definition des membres pour le cas complexe //
236 ////////////////////////////////////////////////
237
238 ...
239
240 template<class T,int ND> FAD<T,ND> FAD< std::
      complex<T>,ND>::norm() const
241 {
242   const T& re = std::real(x);
243   const T& im = std::imag(x);
244   FAD<T,ND> r( std:: sqrt(re*re+im*im),false);
245   for(int i=0;i<ND;i++)
246   {
247     const T& dre = std::real(dx[i]);
248     const T& dim = std:: imag(dx[i]);
249     r.d(i) = (re*dre + im*dim) / r.val();
250   }
251   return r;
252 }
253
254 template<class T,int ND> template<typename X>
255 FAD< std:: complex<T>,ND>::operator FAD<X,ND>() const
256 {
257   FAD<X,ND> res((X) x,false);
258   for(int i=0;i<ND;i++) res.d(i) = (X) dx[i];
259   return res;
260 }
261
262 ...
263
264
265 #endif
```

Table des figures

Bibliographie

[1] David Abrahams et Aleksey Gurtovoy : *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] www.autodiff.org : *The community portal to automatic differentiation*.

[3] R. A. Adams : *Sobolev Spaces*. Academic Press, 1975.

[4] Grégoire Allaire : *Analyse numérique et optimisation*. Editions de l'Ecole Polytechnique, 2005.

[5] Arpack : *Arnoldi Package* - http ://www.caam.rice.edu/software/arpack/.

[6] Aubert, Pierre et Nicolas Di Césaré : Expression templates and forward mode automatic differentiation. *In* George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët et Uwe Naumann, éditeurs : *Automatic Differentiation of Algorithms : From Simulation to Optimization*, Computer and Information Science, chapitre 37, pages 311–315. Springer, New York, NY, 2002.

[7] S. Auliac, Z. Belhachmi, F. Ben Belgacem et F. Hecht : Quadratic finite elements with non-matching grids for the unilateral boundary contact. *ESAIM : Mathematical Modelling and Numerical Analysis*, 2013.

[8] A. K. Aziz : *The mathematical foundations of the finite element method with applications to partial differential equations*. Academic Press, New York, 1972.

[9] L. Baillet et T. Sassi : Mixed finite element formulation in large deformation frictional contact problem. *European Journal of Computational Mechanics*, 14, 2005.

[10] Z. Belhachmi et F. Ben Belgacem : Quadratic finite element for signorini problem. *Math. Comp.*, 72, 2003.

[11] F. Ben Belgacem, P. Hild et P. Laborde : Extension of the mortar finite element method to a variational inequality modeling unilateral contact. *Math. Models Methods Appl. Sc.*, 9, 1999.

[12] F. Ben Belgacem, Y. Renard et L. Slimane : A mixed formulation for the signorini problem in nearly incompressible elasticity. *Applied Numerical Mathematics*, 54, 2005.

[13] C. Bernardi et G. Raugel : Méthodes d'éléments finis mixtes pour les équations de stokes et de navier-stokes dans un polygone non convexe. *Calcolo*, 18:255–291, 1981.

[14] Christine Bernardi, Yvon Maday et Anthony T. Patera : A new non conforming approach to domain decomposition : The mortar element method. *In* Haim Brezis et Jacques-Louis Lions, éditeurs : *Collège de France Seminar*. Pitman, 1994.

[15] E. G. Birgin et J. M. Martínez : Improving ultimate convergence of an augmented lagrangian method. *Optimization Methods Software*, 23(2):177–195, avril 2008.

[16] J. Frédéric Bonnans, Jean Charles Gilbert, Claude Lemaréchal et Claudia A. Sagastizábal : *Numerical Optimization : Theoretical and Practical Aspects (Universitext)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] Houman BOROUCHAKI, Paul Louis GEORGE, Frédéric HECHT, Patrick LAUG et Eric SALTEL : Delaunay mesh generation governed by metric specifications. part i. algorithms. *Finite Elements in Analysis and Design*, 25:61–83, 1997.

[18] S.C. BRENNER et L.R. SCOTT : *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, 2002.

[19] Haïm BREZIS : *Analyse Fonctionnelle*. Dunod, 1999.

[20] F. BREZZI, W. W. HAGER et P. A. RAVIART : Error estimates for the finite element solution of variational inequalities. *Numer. Math.*, 28, 1977.

[21] Franco BREZZI et Michel FORTIN : *Mixed and Hybrid Finite Element Methods*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[22] M. J. CASTRO-DIAZ, F. HECHT, B. MOHAMMADI et O. PIRONNEAU : Anisotropic unstructured mesh adaptation for flow simulations. *International Journal for Numerical Methods in Fluids*, 25:475–491, 1997.

[23] M.J. CASTRO-DIAZ, F. HECHT et B. MOHAMMADI : New progress in anisotropic grid adaptation for inviscid and viscous flows simulations. Rapport technique, INRIA, 1995.

[24] L. CAZABEAU, Y. MADAY et C. LACOUR : Numerical quadratures and mortar methods. *In* Bristeau et AL., éditeur : *Computational sciences for the 21-st Century*. Wiley and Sons, 1997.

[25] I. CHARPENTIER : Checkpointing Schemes for Adjoint Codes : Application to the Meteorological Model Meso-NH. *SIAM J. Sci. Comput.*, 22(6):2135–2151, juin 2000.

[26] Long CHEN, Pengtao SUN et Jinchao XU : Optimal anisotropic meshes for minimizing interpolation errors in L^p-norm, 2006.

[27] A. CHERNOV, M. MAISCHAK et E. P. STEPHAN : *hp*-mortar boundary element method for the two)body contact problems with friction. *Math. Meth. Appl. Sci.*, 31, 2008.

[28] P.-G. CIARLET : *The finite element method for elliptic problems*. North Holland, 1978.

[29] P.-G. CIARLET : *Introduction à l'analyse numérique matricielle et à l'optimisation*. Collection Mathématiques appliquées pour la maîtrise. Masson, 1982.

[30] F.H. CLARKE : *Optimization and Nonsmooth Analysis*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983.

[31] https://www.lri.fr/~hansen/cmaesintro.html - *The CMA Evolution Strategy* .

[32] Claude COHEN-TANNOUDJI, Bernard DIU et Franck LALOÉ : *Mécanique quantique vol. I et II*. Enseignement des sciences. Hermann, Paris, 2000.

[33] Andrew R. CONN, Nicholas I. M. GOULD et Philippe L. TOINT : A globally convergent augmented lagrangian algorithm for optimization with general constraints and simple bounds. *SIAM J. Numer. Anal.*, 28(2):545–572, février 1991.

[34] M. CROUZEIX et V. THOMÉE : The stability in l^p and $w^{1,p}$ of the l^2-projection on finite element function spaces. *Math. Comp.*, 48, 1987.

[35] Ionut DANAILA : Three-dimensional simulations of quantized vortices in rotating bose-einstein condensates, June 30 - July 1 2006.

[36] Ionut DANAILA et Frédéric HECHT : A finite element method with mesh adaptivity for computing vortex states in fast-rotating Bose-Einstein condensates. *Journal of Computational Physics*, 229(19):6946–6960, 2010.

[37] J. E. DENNIS, Jr. et Robert B. SCHNABEL : *Numerical Methods for Unconstrained Optimization and Nonlinear Equations (Classics in Applied Mathematics, 16)*. Soc for Industrial & Applied Math, 1996.

[38] G. DUVAUT et J.-P. LIONS : *Les inéquations en mécanique et en physique*. Dunod, 1972.

[39] Evolving objects.

[40] S. FALETTA : The approximate integration in the mortar method constraint. *In* Lecture notes in computational SCIENCE et ENGINEERING, éditeurs : *Domain decomposition methods in science and engineering XVI*, volume 55-part III. 2007.

[41] R. S. FALK : Error estimates for the approximation of a class of variational inequalities. *Math. of Comp.*, 36, 1974.

[42] K. A. FISCHER et P. WRIGGERS : Frictionless 2d contact formulations for finite deformations based on the mortar method. *Computational Mechanics*, 36, 2005.

[43] B. FLEMISCH, M. A. PUSO et I. WOHLMUTH : A new dual mortar method for curved interfaces : 2d elasticity. *International journal for numerical methods in engineering*, 63, 2005.

[44] Anders FORSGREN, Philip E. GILL et Margaret H. WRIGHT : Interior methods for nonlinear optimization. *SIAM Review*, 44:525–597, 2002.

[45] V. GIRAULT et P.A. RAVIART : *Finite element methods for Navier-Stokes equations : theory and algorithms*. Springer series in computational mathematics. Springer-Verlag, 1986.

[46] Andreas GRIEWANK et Andrea WALTHER : *Evaluating Derivatives : Principles and Techniques of Algorithmic Differentiation*. Numéro 105 de Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd édition, 2008.

[47] Nikolaus HANSEN : The CMA Evolution Strategy : A Comparing Review, 2006.

[48] Laurent HASCOËT et Valérie PASCUAL : TAPENADE 2.1 user's guide. Rapport Technique RT-0300, INRIA, 2004.

[49] J. HASLINGER, I. HLAÁCEK et J. NECAS : Numerical methods for unilateral problems in solid mechanics. *In* P.H. CIARLET et J.L. LIONS, éditeurs : *Handbook of numerical analysis*, volume 4. North Holland, 1996.

[50] Antoine HENROT et Michel PIERRE : *Variation et optimisation de forme*. Springer, 2005.

[51] P. HILD : *Problèmes de contact unilatéral et maillages éléments finis incompatibles*. Thèse de doctorat, Université Paul Sabatier, Toulouse 3, 1998.

[52] P. HILD : Numerical implementation of two nonconforming finite element methods for unilateral contact. *Comput. Methods Appl. Mech. Engrg.*, 2000.

[53] P. HILD et P. LABORDE : Quadratic finite element methods for unilateral contact problems. *Applied Numerical Mathematics*, 41, 2002.

[54] P. HILD et Y. RENARD : An improved a priori error analysis for finite element approximations of signorini's problem. *Siam. J. Numer. Anal.*, 2012.

[55] D. HUA et L. WANG : A mixed finite element method for the unilateral contact problem in elasticity. *Sci. China Ser. A*, 49, 2006.

[56] S. HUEBER, M. MAIR et B.I. WOHLMUTH : A priori error estimates and an inexact primal-dual active set strategy for linear and quadratic finite elements applied to multibody contact problems. *Applied Numerical Mathematics*, 54, 2005.

[57] S. HUEBER et B. I. WOHLMUTH : An optimal a priori error estimate for nonlinear multibody contact problems. *SIAM J. Numer. Anal.*, 43, 2005.

[58] IPOPT home page at the coin-or project website - https://projects.coin-or.org/Ipopt.

[59] Introduction to ipopt : a tutorial for downloading, installing, and using ipopt - *documentation en ligne*.

[60] Steven G. JOHNSON : The nlopt nonlinear-optimization package.

[61] Maarten KEIJZER, J. J. MERELO, G. ROMERO et M. SCHOENAUER : Evolving objects : A general purpose evolutionary computation library. *Artificial Evolution*, 2310:829–888, 2002.

[62] N. KIKUCHI et J. T. ODEN : *Study of variational inequalities and finite element methods*. SIAM, 1988.

[63] T. Y. KIM, J. E. DOLBOW et T. A. LAURSEN : A mortared finite element method for frictional contact on arbitrary surfaces. *Computational Mechanics*, 39, 2007.

[64] W Gouveia L. DENG : The CWP object-oriented optimization library - http ://-coool.mines.edu/, 2003.

[65] T. A. LAURSEN, M. A. PUSO et J. SANDERSC : Mortar contact formulations for deformable-deformable contact : past contributions and new extensions for enriched and embedded interface formulations. *Computer methods in applied mechanics and engineering*, 205-208, 2012.

[66] T. A. LAURSEN et B. YANG : New devlopments in surface-to-surface discretization strategies for analysis of interface mechanics. *Computational Methods in Applied Sciences*, 7, 2012.

[67] M.-X. LI, Q. LIN et S.-H. ZHANG : Superconvergence of finite element method for the signorini problem. *Computational and Applied Mathematics*, 222, 2008.

[68] Elliott H. LIEB et Robert SEIRINGER : Derivation of the gross-pitaevskii equation for rotating bose gases. *Commun. Math. Phys.*, 264:505–537, 2006.

[69] K. W. MADISON, F. CHEVY, W. WOHLLEBEN et J. DALIBARD : Vortex formation in a stirred bose-einstein condensate. *Phys. Rev. Lett.*, 84:806–809, Jan 2000.

[70] Scott MEYERS : *Effective C++ : 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[71] Jean-Marie MIREBEAU : *Approximation adaptive et anisotrope par éléments finis : Théorie et Algorithmes*. Thèse de doctorat, Université Pierre et Marie Curie -Paris 6, 2010.

[72] B. MOHAMMADI et O. PIRONNEAU : *Applied Shape Optimization for Fluids*. Numerical Mathematics and Scientific Computation. OUP Oxford, 2010.

[73] M. MOUSSAOUI et K. KHODJA : Régularité des solutions d'un problème mêlé dirichlet–signorini dans un domaine polygonal plan. *Commun. Part. Diff. Eq.*, 17, 1992.

[74] Uwe NAUMANN : Optimal Jacobian accumulation is NP-complete. *Math. Program.*, 112(2):427–441, 2008.

[75] Uwe NAUMANN : `dcc` User Guide, 2012.

[76] Uwe NAUMANN : *The Art of Differentiating Computer Programs : An Introduction to Algorithmic Differentiation*. SIAM, 2012.

[77] J. NOCEDAL et S. J. WRIGHT : *Numerical Optimization*. Springer-Verlag New York, Inc., New York, 2nd édition, 2006.

[78] Eric T. PHIPPS et Roger P. PAWLOWSKI : Efficient Expression Templates for Operator Overloading-based Automatic Differentiation. *CoRR*, abs/1205.3506, 2012.

[79] Olivier PIRONNEAU, Frédéric HECHT et Jacques MORICE : FreeFEM++, Third Edition, v3.20, www.freefem.org, 2013.

[80] William H. PRESS, Saul A. TEUKOLSKY, William T. VETTERLING et Brian P. FLANNERY : *Numerical Recipes 3rd Edition : The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3rd édition, 2007.

[81] DOLPHIN project-team INRIA LILLE : Paradiseeo.

[82] M. A. PUSO et T. A LAURSEN : A mortar segment-to-segment contact method for large deformation solid mechanics. *Computer methods in applied mechanics and engineering*, 193, 2004.

[83] M. A. PUSO, T. A. LAURSEN et J. SOLBERG : A segment-to-segment mortar contact method for quadratic elements and large deformations. *Comput. Meth. Appl. Mech. and Eng.*, 197, 2008.

[84] Yousef SAAD : *Numerical Methods for Large Eigenvalue Problems*. Manchester University Press, Manchester, UK, 2nd édition, 2011.

[85] K. SCHNEE et J. YNGVASON : Bosons in disc-shaped traps : From 3d to 2d. *Mathematical Physics*, 269:659–691, 2007.

[86] P. SESHAIYER et M. SURI : $h - p$ convergence results for the mortar finite element method. *Math. Comp.*, 69, 2000.

[87] L. SLIMANE : *Méthodes mixtes et traitement du verouillage numérique pour la résolution des inéquations variationnelles*. Thèse de doctorat, Institut national des sciences appliquées de Toulouse, 2001.

[88] Bjarne STROUSTRUP : *The C++ Programming Language*. Addison-Wesley, special édition, 2004.

[89] Bjarne STROUSTRUP : *The C++ Programming Language*. Addison-Wesley, 4th édition, 2013.

[90] Tetgen : A quality tetrahedral mesh generator and a 3d delaunay triangulator.

[91] Jukka I. TOIVANEN et Raino A. E. MAKINEN : Implementation of sparse forward mode automatic differentiation with application to electromagnetic shape optimization. *Optimization Methods Software*, 26(4-5):601–616, octobre 2011.

[92] A. WÄCHTER et L. T. BIEGLER : On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming*, 106(1):25–57, 2006.

[93] Andreas WÄCHTER et Lorenz T. BIEGLER : Line search filter methods for nonlinear programming : Motivation and global convergence. *SIAM J. on Optimization*, 16(1): 1–31, mai 2005.

[94] A. WALTHER et A. GRIEWANK : Getting started with ADOL-C. *In* U. NAUMANN et O. SCHENK, éditeurs : *Combinatorial Scientific Computing*, chapitre 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.

[95] B. I. WHOLMUTH : Monotone multigrid methods on nonmatching grids for nonlinear multibody contact problems. *SIAM journal on scientific computing*, 25, 2003.

[96] B. I. WOHLMUTH : A mortar finite element method using dual spaces for the lagrange multiplier. *SIAM journal on numerical analysis*, 38, 2001.

[97] B YANG, T. A. LAURSEN et X. MENG : Two dimensional mortar contact methods for large deformation frictional sliding. *Internat. J. Numer. Methods Engrg.*, 62, 2005.

[98] Z.-H. ZHONG : Finite element procedures for contact-impact problems. *Oxford University Press*, 1993.